Pedro Miguel Henriques Correia

Problema da Árvore de Suporte de Custo Mínimo com Restrições de Salto

Dissertação apresentada à Universidade de Aveiro para cumprimento dos requisitos necessários à obtenção do grau de Mestre em Matemática e Aplicações, realizada sob a orientação científica da Professora Doutora Maria Cristina Saraiva Requejo Agra, Professora Auxiliar do Departamento de Matemática da Universidade de Aveiro.

o júri / the jury

presidente / president

Doutor Domingos Moreira Cardoso
Professor Catedrático da Universidade de Aveiro (por delegação do Reitor da Universidade de Aveiro)

vogais / examiners committee

Doutor Pedro Martins Pereira Serrão de Moura
Professor Auxiliar da Faculdade de Ciências da Universidade de Lisboa

Doutora Maria Cristina Saraiva Requejo Agra
Professora Auxiliar da Universidade de Aveiro (orientadora)

agradecimentos / acknowledgements

Em primeiro lugar agradeço à Professora Doutora Maria Cristina Saraiva Requejo Agra a forma como orientou a minha dissertação, a liberdade de acção que me permitiu, as suas recomendações e a cordialidade com que sempre me recebeu.

Em segundo lugar agradeço à minha família e em especial aos meus pais por todo o apoio e incentivo ao longo destes dois anos de mestrado.

Agradeço a todos os que contribuíram, de forma directa ou indirecta, para a realização desta dissertação.

palavras-chave

Árvore de Suporte de Custo Mínimo, Restrições de Salto, Algoritmo Genético, Codificação por Sequências de Prüfer, Codificação por Sequências de Arestas, Algoritmo de Prim, Algoritmo PrimRST, Heurística HRST, Algoritmo PrimHMRST

resumo

Nesta dissertação é apresentada uma implementação de um algoritmo genético para o problema da Árvore de Suporte de Custo Mínimo com Restrições de Salto. Este é um problema de optimização combinatória $\mathcal{NP}$-Difícil, associado a problemas de desenho de redes de telecomunicações centralizadas. Nestas redes um dispositivo central deve ser ligado a outros dispositivos periféricos, sem exceder um número máximo de ligações intermédias, designado por salto H, de forma a garantir a integridade e qualidade do sinal da ligação.

O algoritmo genético implementado considera duas codificações para os cromossomas, a codificação por sequências de Prüfer e a codificação por sequências de arestas. A geração da população consiste em dois métodos, um aleatório e um heurístico que considera a restrição de salto do problema.

Os resultados computacionais mostram a performance destes dois métodos de geração da população, assim como a influência de vários parâmetros do algoritmo genético na solução obtida, para as duas codificações em estudo. Os parâmetros considerados são: o número máximo de iterações do algoritmo genético, a dimensão da população, a dimensão de um torneio, o número de torneios, a percentagem de mutação e o número de iterações para renovação da população.

keywords

Minimum Spanning Tree, Hop Constraints, Genetic Algorithm, Prüfer Coding, Edge-Set Coding, Prim Algorithm, PrimRST Algorithm, HRST Heuristic, PrimHMRST Algorithm

abstract

In this thesis we present a genetic algorithm implementation for the Hop-Constrained Minimum Spanning Tree problem (HMST). This is an $\mathcal{NP}$-Hard combinatorial optimization problem, associated with centralized telecommunications network design problems. In these networks a central device must be connected to other devices, without exceeding the maximum number of in-between connections, known as hop H, in order to ensure the signals integrity and quality.

The implemented genetic algorithm considers two chromosome codings, the Prüfer coding and the edge-set coding. Two methods are considered for generating the population, a random and an heuristic method which considers the hop constraints of the problem.

The computational results show the performance of the algorithms for these two methods, as well as the influence of the genetic algorithm parameters in the solutions, considering the two chromosome codings. The studied genetic algorithm parameters are: maximum number of iterations for the genetic algorithm, population dimension, tournament dimension, number of tournaments, mutation percentage and the number of iterations for population renewal.

Conteúdo

Lista de Figuras

Lista de Tabelas

Lista de Algoritmos

Capítulo 1

Introdução

1.1 Objectivos e Âmbito da Dissertação

Nesta dissertação serão estudados os fundamentos dos *algoritmos genéticos* e a sua aplicação ao problema da determinação da *Árvore de Suporte de Custo Mínimo com Restrições de Salto*. Será apresentada uma implementação deste problema recorrendo a duas codificações para os cromossomas do algoritmo genético, nomeadamente a codificação por sequências de Prüfer e a codificação por sequências de arestas. Também será apresentada uma comparação sobre a influência de diferentes parâmetros do algoritmo genético na solução obtida.

O problema da *Árvore de Suporte de Custo Mínimo com Restrições de Salto*, conhecido na literatura inglesa como *Hop-constrained Minimum Spanning Tree problem* ($HMST$), é um problema de optimização combinatória $\mathcal{NP}$-Difícil, que está, geralmente, associado a problemas de desenho de redes de telecomunicações centralizadas [6, 17, 20, 21]. Nestas redes, um dispositivo central deve ser ligado a outros dispositivos periféricos, sem exceder um número máximo de ligações intermédias, designado por salto H, de forma a garantir a integridade e qualidade do sinal da ligação.

Os *algoritmos genéticos* são técnicas evolutivas robustas, que permitem obter soluções ou aproximações a soluções para problemas com os mais diversos graus de complexidade.

Os *algoritmos genéticos* têm aplicabilidade nas mais variadas áreas. Na literatura encontramos algoritmos genéticos aplicados à área do planeamento e afectação, por exemplo, em problemas de produção, inventário e distribuição [27], e em problemas do caixeiro viajante [13]; na área dos sistemas de controlo, temos exemplos de sistemas de controlo de redes de gás e detecção de fugas [10, 11], sistemas de controlo da mistura de ar/gás em

fornalhas [8]; na área da electrónica e inteligência artificial, temos exemplos da integração de chips de alta densidade [36] e na navegação de robôs [24]; entre muitas outras áreas e problemas.

1.2 Trabalhos Anteriores

Na literatura encontramos diversos artigos sobre a aplicação de algoritmos genéticos a problemas com árvores de suporte. Um dos pontos fundamentais da aplicação de algoritmos genéticos a problemas com árvores de suporte, consiste na codificação usada para representar um indivíduo da população. Nos próximos parágrafos são identificadas algumas codificações aplicadas a problemas com árvores de suporte.

Para o problema da *Árvore de Suporte de Custo Mínimo com Restrições de Capacidade* (*Capacitated Minimum Spanning Tree problem*), é usada a codificação por predecessores [33].

Para o problema da *Árvore de Suporte de Custo Mínimo com Restrições de Diâmetro* (*Diameter-constrained Minimum Spanning Tree problem*), é usada a codificação por permutações [22].

Para o problema da *Árvore de Suporte de Custo Mínimo com Restrições de Grau* (*Degree-constrained Minimum Spanning Tree problem*), são usadas as codificações por sequências de inteiros [25], por pesos [23], por sequências de Prüfer [26] e por sequências de arestas [34].

Nesta dissertação serão estudadas estas últimas duas codificações para o problema da *Árvore de Suporte de Custo Mínimo com Restrições de Salto*.

Durante a elaboração deste documento, não tivemos conhecimento sobre trabalhos referentes à aplicação de algoritmos genéticos a problemas com *Árvores de Suporte de Custo Mínimo com Restrições de Salto*. No entanto existe uma referência bibliográfica sobre um problema derivado da *Árvore de Suporte de Custo Mínimo com Restrições de Salto*, identificado pelos autores como *Árvore de Suporte do Fluxo de Custo Mínimo com Restrições de Salto* (*Hop-constrained Minimum cost Flow Spanning Tree problem*), no qual aplicam um algoritmo genético multi-populacional juntamente com um método de pesquisa local [14]. Outros métodos aplicados ao problema da *Árvore de Suporte de Custo Mínimo com Restrições de Salto* consistem em relaxações lagrangeanas [6, 17, 19] e numa heurística simples de construção [17]. Em [21] é apresentado um modelo em programação dinâmica,

juntamente com um algoritmo de pesquisa de soluções em vizinhanças da solução actual. Em [16, 17, 18, 20] são apresentados modelos em programação linear inteira, cuja relaxação linear é usada para aproximar o valor do problema.

1.3 Estrutura da Dissertação

Esta dissertação está organizada em oito capítulos, um glossário, três apêndices e a respectiva bibliografia.

Neste primeiro capítulo é feita a descrição dos objectivos e âmbito da dissertação, são referidos alguns exemplos da aplicação dos algoritmos genéticos a diversas áreas e problemas, são referidos trabalhos sobre árvores de suporte, com incidência nas codificações de cromossomas usadas, assim como a presente descrição dos capítulos.

No segundo capítulo é feita uma pequena introdução à teoria de grafos com especial destaque para as árvores de suporte. Para a determinação de uma *Árvore de Suporte de Custo Mínimo*, é apresentado e exemplificado o *Algoritmo de Prim*. Posteriormente é descrito o problema da determinação da *Árvore de Suporte de Custo Mínimo com Restrições de Salto*.

No terceiro capítulo é abordada a temática dos algoritmos genéticos, onde são descritos os elementos que constituem os cromossomas, a avaliação da admissibilidade e cálculo da aptidão dos mesmos, a constituição da população, e o processo evolutivo, através da selecção, cruzamento, mutação, renovação e elitismo. São descritos os métodos de cruzamento baseados no *K-Point Crossover* e no *Uniform Crossover*, e o método de mutação *Single Point Mutation*.

No quarto capítulo são aprofundadas as noções sobre o conceito de codificação associado aos algoritmos genéticos e árvores de suporte, com incidência na codificação de cromossomas por sequências de Prüfer e por sequências de arestas. Para a codificação por sequências de arestas, são apresentados outros métodos de cruzamento e mutação, com especial incidência no método *PrimRST* que foi introduzido em [34].

No quinto capítulo é descrita a implementação de um algoritmo genético para o problema da determinação da *Árvore de Suporte de Custo Mínimo com Restrições de Salto* usando as duas codificações de cromossomas apresentadas no quarto capítulo. Neste capítulo é apresentada uma heurística que permite gerar indivíduos que respeitam o valor da restrição de salto. É apresentado o algoritmo do método de cruzamento introduzido no

capítulo anterior, *PrimRST*.

No sexto capítulo são apresentados os resultados computacionais obtidos usando o algoritmo genético, descrito no quinto capítulo, para obter soluções para diferentes concretizações de problemas e a comparação da influência dos parâmetros do algoritmo genético.

No sétimo capítulo é apresentado um algoritmo baseado no método *PrimRST* que respeita o valor da restrição de salto, permitindo gerar indivíduos admissíveis para cada operação de cruzamento.

No oitavo capítulo são realizadas as considerações finais sobre esta dissertação e referidos alguns tópicos para trabalhos futuros.

Após o oitavo capítulo é apresentado um glossário, com os termos usados nas tabelas apresentadas no sexto capítulo e no primeiro apêndice.

O primeiro apêndice apresenta as tabelas com resultados computacionais que não foram colocadas no sexto capítulo.

O segundo apêndice apresenta a documentação do algoritmo genético implementado em Java, para a codificação de cromossomas por sequências de Prüfer.

O terceiro apêndice apresenta a documentação do algoritmo genético implementado em Java, para a codificação de cromossomas por sequências de arestas.

Por fim, é apresentada a bibliografia usada na elaboração desta dissertação.

Capítulo 2

Árvores de Suporte

2.1 Noções de Grafos

Nesta secção será feita uma pequena introdução à teoria dos grafos cujo objectivo será o de apresentar alguns conceitos que serão utilizados nas próximas secções e capítulos desta dissertação. Todos os conceitos apresentados nesta secção são baseados em [4]. Comecemos pela definição de grafo.

Definição 2.1 (Grafo)**.**
Um grafo G, representado por $G = (V, E)$, é constituído por um conjunto finito e não vazio de vértices, denotado por $V = V(G) = \{v_1, v_2, \ldots, v_n\}$, e por um conjunto finito de arestas, denotado por $E = E(G) = \{e_1, e_2, ..., e_m\}$.

Uma aresta $e \in E(G)$ pode ser representada por um par não ordenado de vértices $u, v \in V(G)$, $e = \{u, v\} = \{v, u\}$. A ligação entre esse par de vértices, indica que a aresta e é *incidente* nos vértices u e v. Por outro lado, os vértices u, v são vértices extremos da aresta e, logo dizem-se *vértices adjacentes*.

Uma aresta representada por um par não ordenado de vértices não distintos é designado por *lacete*, enquanto que duas ou mais arestas representadas por um mesmo par não ordenado de vértices distintos são designadas por *arestas paralelas*.

Definição 2.2 (Ordem e Dimensão de um grafo)**.**
A cardinalidade do conjunto V representa o número de vértices no grafo G, sendo designada por ordem *de G e denotada por $|V(G)| = n$.*

A cardinalidade do conjunto E representa o número de arestas no grafo G, sendo designada por dimensão *de G e denotada por* $|E(G)| = m$.

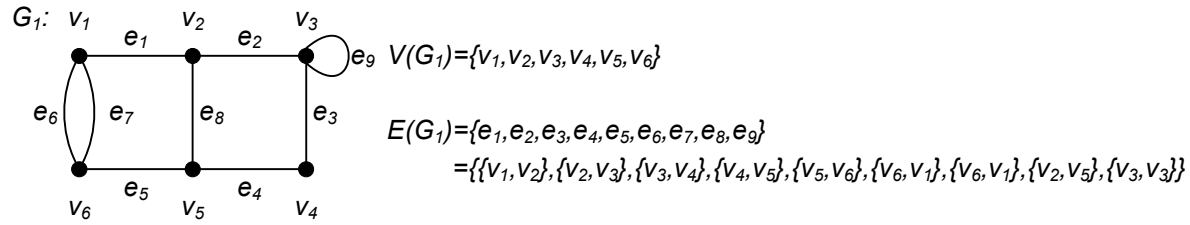

Figura 2.1: Representação de um grafo com lacetes e arestas paralelas.

O grafo apresentado na Figura 2.1, G_1, exemplifica um grafo com um *lacete* e com *arestas paralelas*, tendo $|V(G_1)| = 6$ vértices e $|E(G_1)| = 9$ arestas, logo dizemos que G_1 tem ordem 6 e dimensão 9.

Designamos por *grafo simples*, um grafo que não contém lacetes nem arestas paralelas.

Definição 2.3 (Grau de um vértice)**.**
O grau de um vértice $v \in V(G)$ *consiste no número de arestas incidentes em* v*, e denota-se por* $d_G(v)$.

No caso de um vértice com lacetes, cada um dos lacetes conta como duas incidências.

Definição 2.4 (Vizinhança de um vértice)**.**
A vizinhança de um vértice $v \in V(G)$ *consiste no conjunto de vértices adjacentes a* v*, e denota-se por* $N_G(v)$.

Considerando o grafo da Figura 2.1, o grau do vértice v_3 é dado por $d_{G_1}(v_3) = 4$, que corresponde à incidência de duas arestas e_2 e e_3, e ao lacete e_9, e a vizinhança do vértice v_2 é dada por $N_{G_1}(v_2) = \{v_1, v_3, v_5\}$.

Definição 2.5 (Passeio)**.**
Consideremos dois vértices $u, v \in V(G)$*. Um* passeio *entre os vértices* u *e* v*, é uma sequência finita e alternada de vértices e arestas, com eventual repetição de vértices e arestas, dada por* $P = v_0 e_1 v_1 e_2 \dots e_k v_k$*, no qual a aresta* e_i *é incidente nos vértices* v_{i-1} *e* v_i*, e os vértices* u *e* v *correspondem a* v_0 *e* v_k*, respectivamente.*

Se os vértices inicial e final do passeio forem distintos, temos um *passeio aberto*, caso contrário, temos um *passeio fechado*.

A Definição 2.6 permite expandir a Definição 2.5 e pormenorizar diferentes tipos de passeios em grafos.

Definição 2.6 (Caminho e Ciclo)**.**
Designamos por caminho *um passeio P onde não existem vértices repetidos.*
Um caminho fechado é designado por ciclo*.*

A Figura 2.2 apresenta dois passeios, P_1 e P_2, que exemplificam ambos os conceitos apresentados na Definição 2.6.

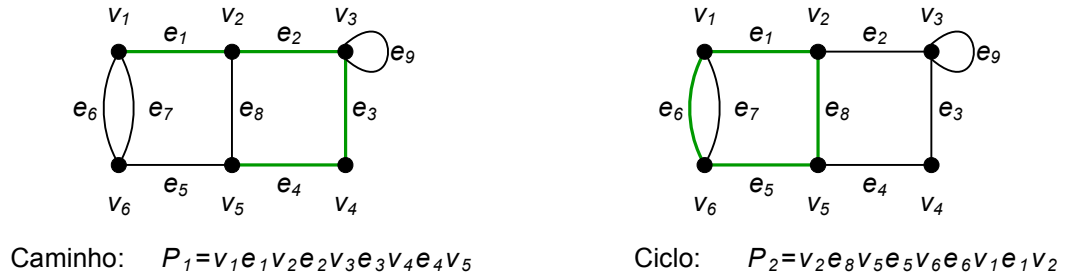

Figura 2.2: Representação de um caminho e de um ciclo.

Definição 2.7 (Comprimento e Distância)**.**
O comprimento *do passeio P, $comp_G(P)$, é dado pelo número de arestas que constituem o passeio P no grafo G.*
A distância *entre dois vértices distintos $u, v \in V(G)$ corresponde ao menor (valor) dos comprimentos de todos os caminhos entre os vértices u e v e representamos por $dist_G(u, v)$. Se não existir um caminho entre u e v, a distância é representada por $dist_G(u, v) = \infty$.*

A Definição 2.7 apresenta dois conceitos métricos associados a grafos. O primeiro refere o comprimento de um caminho P e o segundo refere a distância entre dois vértices de um grafo G. Os comprimentos dos passeios apresentados na Figura 2.2 são os seguintes: $comp(P_1) = comp(P_2) = 4$. A distância entre o vértice v_1 e o vértice v_3 do grafo G_1, apresentado na Figura 2.1, corresponde ao caminho de menor comprimento entre esses vértices, que é $P = v_1e_1v_2e_2v_3$ e no qual o comprimento é $comp_{G_1}(P) = 2$, logo $dist_{G_1}(v_1, v_3) = 2$.

Definição 2.8 (Grafo Conexo)**.**
Dois vértices distintos $u, v \in V(G)$ dizem-se conexos se o grafo G contém pelo menos um caminho entre ambos os vértices. Consequentemente, um grafo G diz-se conexo *se cada par de vértices do grafo for conexo.*

Um grafo simples em que todos os pares de vértices distintos são adjacentes, diz-se *grafo completo* de ordem n e representamos por K_n. Na Figura 2.3 estão representados os cinco primeiros grafos completos, K_1 a K_5.

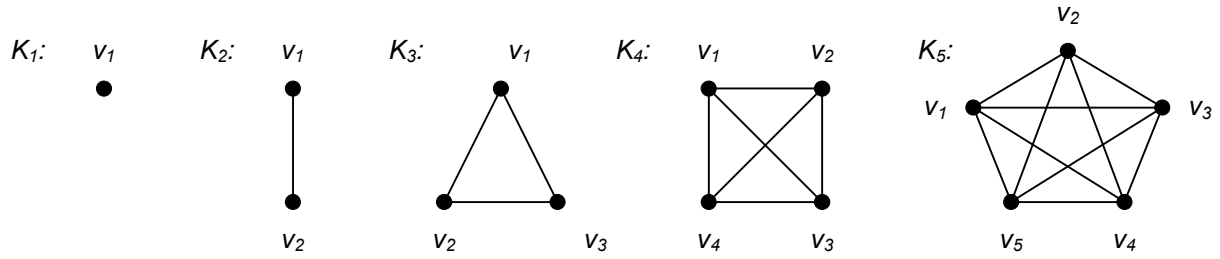

Figura 2.3: Representação dos cinco primeiros grafos completos.

Um grafo H é um *subgrafo* do grafo G se $V(H) \subseteq V(G)$ e $E(H) \subseteq E(G)$, e denotamos por $H \subseteq G$. Se $V(H) = V(G)$ então o subgrafo H contém os mesmos vértices que o grafo G, e designamos por *subgrafo de suporte*.

Definição 2.9 (Árvore)**.**
Uma árvore *T, representada por $T = (V, E)$, é um grafo simples, conexo e sem ciclos.*

Uma *árvore etiquetada* é uma árvore em que os vértices são etiquetados com números inteiros entre 1 e n, sendo n a ordem da árvore. A Figura 2.4 exemplifica uma árvore etiquetada, T_1. Note-se que a árvore T_1 é um subgrafo de suporte do grafo G_1 apresentado na Figura 2.1.

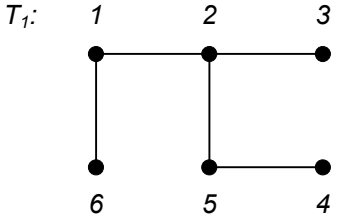

Figura 2.4: Representação de uma árvore etiquetada.

2.2 Árvore de Suporte de Custo Mínimo

As *árvores de suporte* estão, geralmente, associadas a problemas de planeamento de redes informáticas, de televisão, de telefone, de abastecimento de água, luz, gás, e redes

de transportes rodoviários, ferroviários, marítimos e aéreos. Nestas redes, frequentemente pretendemos ligar vários pontos, minimizando o custo ou o comprimento dessas ligações, resultando no problema da *Árvore de Suporte de Custo Mínimo*.

Matematicamente define-se uma árvore de suporte (abrangente ou geradora) como se segue na Definição 2.10. Tal como na secção precedente, as definições são baseadas em [4].

Definição 2.10 (Árvore de Suporte)**.**
Uma árvore de suporte T *de um grafo conexo* G*, é um subgrafo de suporte de* G*, simples, conexo e sem ciclos.*

Por outras palavras, uma *árvore de suporte* T de um grafo conexo G, é uma árvore que tem todos os vértices de G e $|V(G)| - 1$ arestas de G. Nestas condições, qualquer grafo G tem pelo menos uma árvore de suporte, porque sendo G um grafo conexo, existe um caminho entre cada par de vértices de G, e no caso de existirem ciclos, basta remover arestas de modo a que o grafo resultante continue a ser conexo. Por outro lado, o número máximo de árvores de suporte de um grafo G é dado pela *Fórmula de Cayley*, apresentada na Secção 4.2, quando G é um grafo completo.

A Definição 2.11 introduz a notação para o conjunto das árvores de suporte de um grafo G e a sua cardinalidade.

Definição 2.11.
O conjunto de todas as árvores de suporte de um grafo G *é denotado por* $\mathcal{T}(G)$*.*
A cardinalidade deste conjunto corresponde ao número de árvores de suporte do grafo G *e denotamos por* $\tau(G)$*.*

O problema da determinação da *Árvore de Suporte de Custo Mínimo* (*Minimum Spanning Tree problem*) de um grafo G, é um problema de optimização combinatória, tais como os problemas do caixeiro viajante (*Traveling-Salesman problem*), do saco mochila (*Knapsack problem*), de empacotamento (*Bin-Packing problem*), de cobertura de conjuntos (*Set-Covering problem*), de transporte (*Vehicle Routing problem*), entre outros. Estes problemas caracterizam-se por terem um conjunto finito, mas grande, de soluções admissíveis, que no caso do problema da *Árvore de Suporte de Custo Mínimo* é representado por $\mathcal{T}(G)$.

Consideremos uma extensão da Definição 2.1 com a introdução de custos nas arestas, onde o grafo é dado por $G = (V, E, W)$, onde W representa o conjunto dos custos

associados às arestas $e \in E(G)$ e denotamos por $W = W(G) = \{w_1, w_2, \ldots, w_m\}$ com $w_i > 0$, $i = 1, \ldots, m$. Assim a cardinalidade de $W(G)$ é igual à cardinalidade de $E(G)$, $|W(G)| = |E(G)| = m$, e w_e é o custo associado à aresta $e \in E(G)$.

Definição 2.12 (Árvore de Suporte de Custo Mínimo)**.**
Seja G um grafo com custos nas arestas. A correspondente Árvore de Suporte de Custo Mínimo*, T, é uma das árvores de suporte contidas no conjunto $\mathcal{T}(G)$, na qual o somatório dos custos associados às suas arestas, $\sum_{e \in E(T)} w_e$, é o menor entre todas as árvores de suporte.*

Note-se que não existe necessariamente apenas uma árvore de suporte para um grafo. Se considerarmos um grafo com os mesmos custos em todas as arestas, todas as árvores de suporte deste grafo têm o mesmo custo, sendo todas árvores de suporte de custo mínimo.

O primeiro algoritmo para a determinação da *Árvore de Suporte de Custo Mínimo* [38], surgiu em 1926 por Otakar Borůvka, no planeamento de uma rede eléctrica da região da Morávia na República Checa. Este algoritmo é conhecido como *Algoritmo de Borůvka* ou *Algoritmo de Sollin*.

Em 1930 foi apresentado o segundo algoritmo, por Vojtěch Jarník, que mais tarde, em 1957, viria a ser redescoberto por Robert Prim e designado por *Algoritmo de Prim* [38].

Em 1956 foi apresentado o terceiro algoritmo por Joseph Kruskal e designado por *Algoritmo de Kruskal* [38]. Estes são os três algoritmos mais conhecidos para a determinação da *Árvore de Suporte de Custo Mínimo*.

Neste documento será apresentado o *Algoritmo de Prim* na Secção 2.2.1, enquanto que os restantes podem ser consultados em [1].

O problema da determinação da *Árvore de Suporte de Custo Mínimo* T pode ser formulado como um problema de programação linear. Consideremos um grafo $G = (V, E, W)$ e uma notação mais abreviada, onde w_{ij} é equivalente a $w_{\{i,j\}}$ com $\{i, j\} \in E$. Consideremos, ainda, uma variável binária x_{ij} na qual:

$$x_{ij} = \begin{cases} 1 & \text{se a aresta } \{i,j\} \in E \text{ pertence à árvore } T \\ 0 & \text{se a aresta } \{i,j\} \in E \text{ não pertence à árvore } T \end{cases}$$

A formulação como um problema de programação linear para a *Árvore de Suporte de Custo Mínimo* T é a seguinte [1]:

$$\min \sum_{\{i,j\}\in E} w_{ij}x_{ij} \tag{2.1}$$

$$\text{s.a} \sum_{\{i,j\}\in E} x_{ij} = |V| - 1 \tag{2.2}$$

$$\sum_{\{i,j\}\in E\,:\,i,j\in S} x_{ij} \leq |S| - 1 \quad \forall S \subseteq V,\ |S| \geq 2 \tag{2.3}$$

$$x_{ij} \in \{0,1\} \qquad\qquad \{i,j\} \in E \tag{2.4}$$

A função objectivo, expressão (2.1), representa o somatório dos custos das arestas na árvore suporte T. A restrição (2.2) indica que a árvore de suporte T tem $|V| - 1$ arestas, estando em concordância com a Definição 2.10. A restrição (2.3) permite eliminar ciclos em subconjuntos de vértices $S \subseteq V$. Por fim temos as restrições de integralidade (2.4).

Os problemas com árvores de suporte, por vezes, requerem a inclusão de uma ou mais restrições. Consideremos o exemplo da implementação de uma rede informática.

- A extensão da rede está limitada à tecnologia de cabo utilizada, variando entre dezenas a centenas de metros para cabos de cobre, e dezenas a centenas de quilómetros para fibra óptica.
- O número de dispositivos ligados a um nó depende do número de ligações suportadas pelo switch/router. Neste caso podemos identificar o problema da *Árvore de Suporte de Custo Mínimo com Restrições de Grau* (*Degree-constrained Minimum Spanning Tree problem*).
- Numa rede wireless os *access points* permitem ampliar a extensão da rede, repetindo o sinal entre um ou mais *access points* até aos dispositivos móveis. Se o número de *access points* entre um servidor e os dispositivos móveis for restringido, então podemos identificar o problema da *Árvore de Suporte de Custo Mínimo com Restrições de Salto* (*Hop-constrained Minimum Spanning Tree problem*).

Os problemas de árvores de suporte de custo mínimo com restrições adicionais, apresentadas como exemplo, podem ser aplicadas a outros problemas.

O problema da *Árvore de Suporte de Custo Mínimo com Restrições de Salto* é aprofundado na Secção 2.3 deste trabalho.

2.2.1 Algoritmo de Prim

O *Algoritmo de Prim* [38] é um algoritmo guloso (*greedy*) porque constrói a solução aresta a aresta, escolhendo sempre a aresta que oferece o melhor benefício imediato. Neste caso é construída uma árvore de suporte T, a partir de um grafo conexo G, onde em cada iteração do algoritmo é adicionada a aresta de menor custo, entre todas as arestas possíveis de adicionar. A árvore T começa com um dos vértices do grafo G e em cada iteração é adicionada a aresta de menor custo que liga um dos vértices da árvore T, $V(T)$, e um dos restantes vértices do grafo G, $V(G) \setminus V(T)$. Este processo evita a formação de ciclos, termina quando a árvore T tem a mesma ordem do grafo G e caracteriza o *Algoritmo de Prim* como um algoritmo guloso.

O Algoritmo 2.1 [38] apresenta os passos executados pelo *Algoritmo de Prim* na determinação da *Árvore de Suporte de Custo Mínimo* T de um grafo G e está exemplificado no Exemplo 2.1.

Algoritmo 2.1: Algoritmo de Prim.

Entrada: $G = (V, E, W) \rightarrow$ grafo conexo com custos nas arestas
Saída: $T = (C, F) \rightarrow$ árvore de suporte
Variáveis: $F \rightarrow$ conjunto das arestas da árvore de suporte T
$C \rightarrow$ conjunto dos vértices da árvore de suporte T
$s, u, v \rightarrow$ vértices temporários

```
1 F ← {}
2 escolher aleatoriamente um vértice inicial s ∈ V
3 C ← {s}
4 enquanto C ≠ V fazer
5 |   escolher a aresta {u, v} de menor custo com u ∈ C e v ∈ V \ C
6 |   F ← F ∪ {u, v}
7 |   C ← C ∪ v
8 devolver T = (C, F)
```

Exemplo 2.1.

Aplicação do Algoritmo 2.1 na determinação da Árvore de Suporte de Custo Mínimo, *T, do grafo G_2 representado na Figura 2.5 e com custos nas arestas.*

Na Tabela 2.1 são apresentados os passos descritos no Algoritmo 2.1 enquanto que a Figura 2.6 representa graficamente os passos executados na Tabela 2.1.

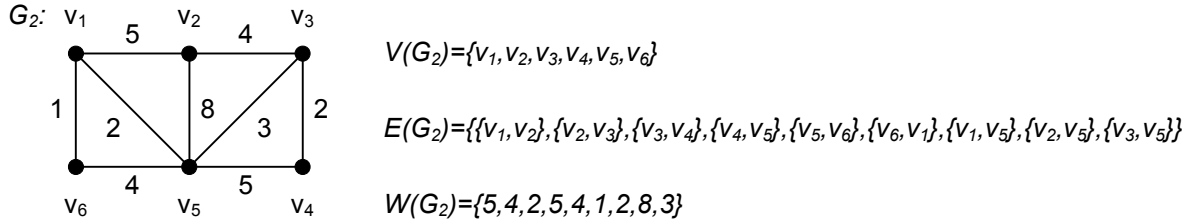

Figura 2.5: Representação do grafo do Exemplo 2.1.

Início	$F \leftarrow \{\}\,,\ s \leftarrow v_4,\ C \leftarrow \{v_4\}$
Iteração 1	$V \setminus C \leftarrow \{v_1, v_2, v_3, v_5, v_6\}\,,\ \{u, v\} \leftarrow \{v_4, v_3\}\,,\ C \leftarrow \{v_4, v_3\}\,,$ $F \leftarrow \{\{v_4, v_3\}\}$
Iteração 2	$V \setminus C \leftarrow \{v_1, v_2, v_5, v_6\}\,,\ \{u, v\} \leftarrow \{v_3, v_5\}\,,\ C \leftarrow \{v_4, v_3, v_5\}\,,$ $F \leftarrow \{\{v_4, v_3\}, \{v_3, v_5\}\}$
Iteração 3	$V \setminus C \leftarrow \{v_1, v_2, v_6\}\,,\ \{u, v\} \leftarrow \{v_5, v_1\}\,,\ C \leftarrow \{v_4, v_3, v_5, v_1\}\,,$ $F \leftarrow \{\{v_4, v_3\}, \{v_3, v_5\}, \{v_5, v_1\}\}$
Iteração 4	$V \setminus C \leftarrow \{v_2, v_6\}\,,\ \{u, v\} \leftarrow \{v_1, v_6\}\,,\ C \leftarrow \{v_4, v_3, v_5, v_1, v_6\}\,,$ $F \leftarrow \{\{v_4, v_3\}, \{v_3, v_5\}, \{v_5, v_1\}, \{v_1, v_6\}\}$
Iteração 5	$V \setminus C \leftarrow \{v_2\}\,,\ \{u, v\} \leftarrow \{v_3, v_2\}\,,\ C \leftarrow \{v_4, v_3, v_5, v_1, v_6, v_2\}\,,$ $F \leftarrow \{\{v_4, v_3\}, \{v_3, v_5\}, \{v_5, v_1\}, \{v_1, v_6\}, \{v_3, v_2\}\}$
Fim	$T = (C, F)$

Tabela 2.1: Representação dos passos descritos no Algoritmo 2.1 para o grafo da Figura 2.5.

Na Figura 2.6, as arestas a preto são as arestas que podem ser incluídas na árvore, as arestas a verde representam a construção da árvore de suporte T, e as arestas vermelhas representam arestas inadmissíveis. Facilmente verificamos que em cada iteração é adicionada uma aresta verde à árvore T e que essa aresta é a de custo mínimo que liga vértices de T e vértices de G que ainda não pertencem a T. Após adicionar a aresta a verde, verificamos que uma ou mais arestas deixam de poder ser consideradas, pois se adicionadas formam ciclos, tornando-se inadmissíveis e sendo representadas a vermelho.

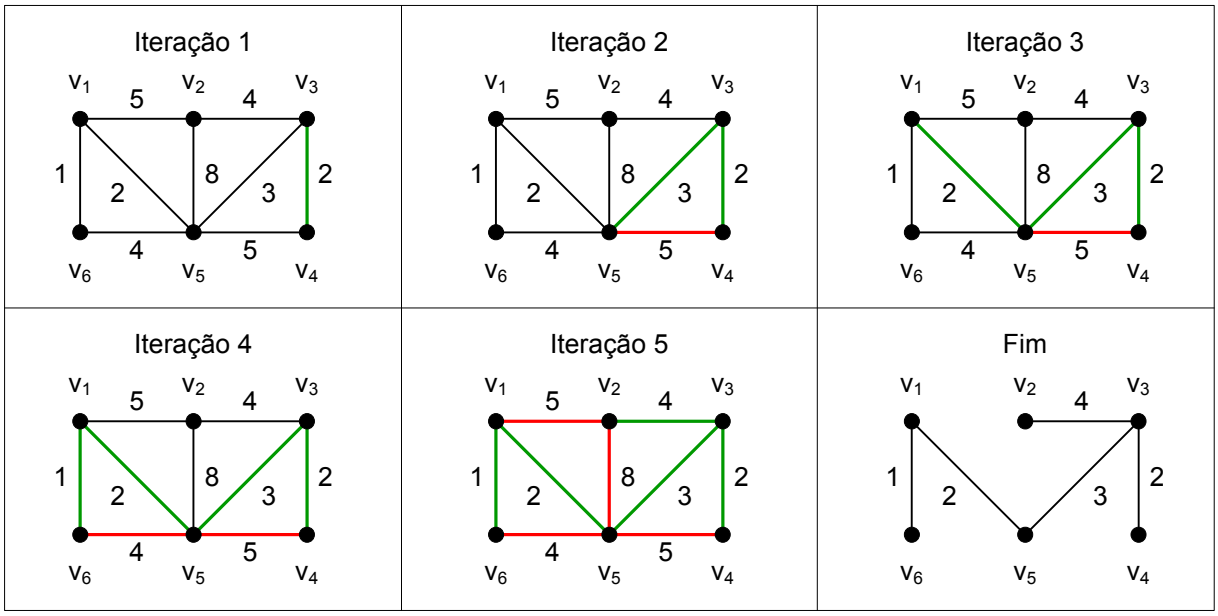

Figura 2.6: Representação gráfica dos passos executados na Tabela 2.1.

□

2.3 Árvore de Suporte de Custo Mínimo com Restrições de Salto

Comecemos por definir o conceito de *salto* associado às árvores de suporte com restrições de salto [21].

Definição 2.13 (Salto)**.**
O salto *entre dois vértices de uma árvore T define-se como a distância entre esses vértices.*

Note-se que a distância entre dois vértices corresponde ao número de arestas do único caminho existente entre esses dois vértices na árvore. Fixando um dos vértices, podemos estabelecer *níveis* [21], que representam a distância entre o vértice fixado e os restantes vértices da árvore. O vértice fixado designa-se por *raiz*, denota-se pela etiqueta v_0 e o seu nível corresponde ao nível 0.

A Figura 2.7 exemplifica uma árvore de suporte, T_2, restringida a três saltos a partir do vértice raiz, v_0. O comprimento do único caminho entre o vértice raiz, v_0, e o vértice v_4 é de duas arestas, logo o vértice v_4 está no nível 2 ou está a 2 saltos da raiz.

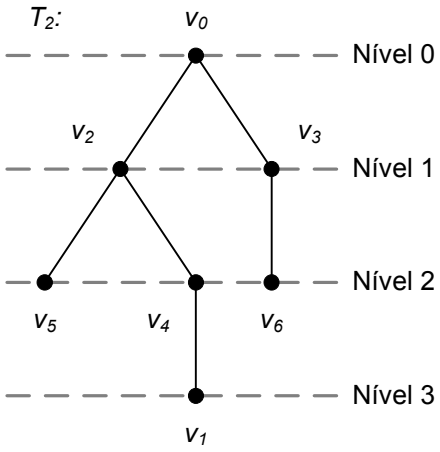

Figura 2.7: Representação gráfica de uma árvore de suporte restringida a três saltos.

O problema da determinação da *Árvore de Suporte T de Custo Mínimo com Restrições de Salto*, também, pode ser formulado como um problema de programação linear. Uma formulação para este problema é proposta por Gouveia [17] e designa-se por *Undirected Multicommodity Flow Formulation* (*UMCF*). Esta formulação é baseada na formulação da *Árvore de Suporte de Custo Mínimo*, apresentada na Secção 2.2, com a introdução de fluxos e das restrições de salto.

Consideremos agora um grafo $G = (V, E, W)$, com $V = \{v_0, v_1, \ldots, v_n\}$ e a mesma notação abreviada, onde w_{ij} é equivalente a $w_{\{i,j\}}$ com $\{i, j\} \in E$. Do conjunto de vértices V, seleccionamos um vértice *raiz*, o vértice v_0. A ordem de G é $|V| = n + 1$, onde n corresponde ao índice do último vértice e, consequentemente a árvore T tem dimensão n.

Consideremos o valor $H \in \mathbb{N}$ (*Hop*), que indica o salto máximo permitido entre o vértice raiz, v_0, e qualquer outro vértice da árvore T.

Por fim, definimos as variáveis binárias, x_{ij}, y_{ij}^k e y_{ji}^k para todo $\{i, j\} \in E$ e $k = 1, \ldots, n$, como se seguem:

$$x_{ij} = \begin{cases} 1 & \text{se a aresta } \{i,j\} \in E \text{ está na árvore } T \\ 0 & \text{se a aresta } \{i,j\} \in E \text{ não está na árvore } T \end{cases}$$

$$y_{ij}^k = \begin{cases} 1 & \text{se a aresta } \{i,j\} \in E \text{ é usada na direcção de } v_i \text{ para } v_j \text{ no caminho entre a raiz e o vértice } v_k \\ 0 & \text{se a aresta } \{i,j\} \in E \text{ não é usada na direcção de } v_i \text{ para } v_j \text{ no caminho entre a raiz e o vértice } v_k \end{cases}$$

$$y_{ji}^k = \begin{cases} 1 & \text{se a aresta } \{i,j\} \in E \text{ é usada na direcção de } v_j \text{ para } v_i \text{ no caminho entre a raiz e o vértice } v_k \\ 0 & \text{se a aresta } \{i,j\} \in E \text{ não é usada na direcção de } v_j \text{ para } v_i \text{ no caminho entre a raiz e o vértice } v_k \end{cases}$$

A variável y_{ij}^k define o fluxo que passa na aresta $\{i,j\}$, na direcção do vértice v_i para o vértice v_j, no caminho entre o vértice raiz, v_0 e o vértice v_k. A variável y_{ji}^k define o fluxo que passa na aresta $\{i,j\}$, na direcção do vértice v_j para o vértice v_i, no caminho entre o vértice raiz, v_0 e o vértice v_k.

A formulação como um problema de programação linear para a *Árvore de Suporte de Custo Mínimo com Restrições de Salto* é a seguinte (formulação *UMCF* [17]):

$$\min \sum_{\{i,j\}\in E} w_{ij}\, x_{ij} \tag{2.5}$$

$$\text{s.a} \sum_{\{i,j\}\in E} x_{ij} = n \tag{2.6}$$

$$\sum_{j=1}^{n} y_{0j}^k = 1 \qquad k = 1,\ldots,n \tag{2.7}$$

$$\sum_{i=0}^{n} y_{ik}^k = 1 \qquad k = 1,\ldots,n \tag{2.8}$$

$$\sum_{i=0}^{n} y_{ij}^k - \sum_{i=1}^{n} y_{ji}^k = 0 \quad j,k = 1,\ldots,n;\ j \neq k \tag{2.9}$$

$$y_{0j}^k \leq x_{0j} \qquad j,k = 1,\ldots,n \tag{2.10}$$

$$y_{ij}^k + y_{ji}^h \leq x_{ij} \qquad \{i,j\} \in E;\ k,h = 1,\ldots,n;\ k \neq i;\ h \neq j \tag{2.11}$$

$$\sum_{\{i,j\}\in E} y_{ij}^k \leq H \qquad k = 1,\ldots,n \tag{2.12}$$

$$y_{ij}^k, y_{ji}^k \in \{0,1\} \qquad \{i,j\} \in E;\ k = 1,\ldots,n \tag{2.13}$$

$$x_{ij} \in \{0,1\} \qquad \{i,j\} \in E \tag{2.14}$$

A expressão (2.5) representa a função objectivo do problema e é semelhante à expressão (2.1). A restrição (2.6) indica que a árvore de suporte T tem n arestas. As restrições (2.7) indicam que o vértice raiz, v_0, envia uma unidade de fluxo para cada vértice v_k, com $k = 1,\ldots,n$. As restrições (2.8) indicam que cada vértice v_k recebe uma unidade de fluxo. As restrições (2.9) garantem a conservação do fluxo nos restantes vértices. As restri-

ções (2.10) indicam que só existe passagem de fluxo numa determinada aresta incidente na raiz, se esta aresta pertencer à *Árvore de Suporte de Custo Mínimo*. As restrições (2.11) indicam que se uma aresta pertence à *Árvore de Suporte de Custo Mínimo* então é usada apenas num dos sentidos dos fluxos dessa aresta. As restrições de salto são implementadas em (2.12), no qual o número de arestas no caminho entre a raiz e o vértice v_k não pode exceder o salto H, para todo $k = 1, \ldots, n$. Por fim, temos as restrições de integralidade para as variáveis y_{ij}^k e y_{ji}^k em (2.13), e para as variáveis x_{ij} em (2.14).

Note-se que, para simplificar a escrita da formulação, não foram consideradas as variáveis y_{ki}^k, com $i, k = 1, \ldots, n;\ i \neq k$, que correspondem ao fluxo que passa na aresta $\{k, i\}$, na direcção do vértice v_k para o vértice v_i, no caminho entre o vértice raiz, v_0 e o vértice v_k.

Como referido na Introdução, Secção 1.1, o problema da determinação da *Árvore de Suporte de Custo Mínimo com Restrições de Salto*, é um problema de optimização combinatória $\mathcal{NP}$-Difícil, tendo como caso particular o *Uncapacitated Simple Facility Location problem* ($H = 2$) que é conhecido como *$\mathcal{NP}$-Difícil* [17], pois não se conhecem algoritmos de tempo polinomial capazes de resolver todas as suas instâncias. Na literatura apresentada na Secção 1.2 foram referidos alguns métodos para resolução deste problema. Nesta dissertação, o problema será resolvido recorrendo a um algoritmo genético que obtém soluções aproximadas para o problema e que correspondem a limites superiores do seu valor óptimo.

No Capítulo 3 são apresentados os moldes dos algoritmos genéticos, que posteriormente são adaptados a este problema nos capítulos seguintes.

Capítulo 3

Algoritmos Genéticos

Os algoritmos genéticos [5, 9, 28, 31, 37] são métodos de optimização inspirados na teoria da evolução por selecção natural e sobrevivência do mais apto, preconizada por Charles Darwin, em 1859, no livro *On the Origin of Species by Means of Natural Selection, or the Preservation of Favoured Races in the Struggle for Life* [7].

Na década de 50, começaram a ser estudadas técnicas evolutivas que permitiam obter soluções para problemas relacionados com inteligência artificial e *machine learning* [28].

Na década de 60, John Holland da Universidade do Michigan, apresentou os moldes dos algoritmos genéticos, mais tarde publicados em *Adaptation in Natural and Artificial Systems*. Em vez de utilizar uma técnica evolutiva específica para um problema específico, criou um algoritmo genérico que imitava os fenómenos de adaptação que ocorrem na natureza, obtendo, deste modo, um método geral, capaz de ser aplicado a uma grande variedade de problemas [28]. Estes moldes foram continuamente desenvolvidos por vários investigadores e posteriormente popularizados por David Goldberg na sua dissertação *Computer-Aided Gas Pipeline Operation using Genetic Algorithms and Rule Learning* [10] e no seu livro *Genetic Algorithms in Search, Optimization and Machine Learning* [11].

A teoria da selecção natural descrita por Charles Darwin, propõe que os animais e plantas que actualmente existem, provêm da adaptação e competição por recursos nos respectivos ecossistemas, durante milhões de anos [37]. Esta atitude de adaptação e competição, existente na natureza, ao longo dos anos, constitui a base evolutiva pelo qual se regem os algoritmos genéticos.

Este processo evolutivo, para os algoritmos genéticos, consiste [5]:

1. num conjunto de indivíduos que constituem uma *população*
2. num método que permite avaliar a *aptidão* dos indivíduos
3. num método que permite combinar indivíduos para obter novos indivíduos
4. num operador de *mutação* para garantir a diversidade da população

Os algoritmos genéticos têm uma terminologia análoga àquela usada pelos biólogos [5]. As soluções admissíveis para um problema, num algoritmo genético, são identificadas por *cromossomas*. Estes cromossomas são compostos por uma *string* ou vector de dígitos. Nem sempre esta *string* de dígitos pode ser directamente aplicada ao problema, pois, geralmente, possui uma codificação diferente. Isto significa que o cromossoma tem de ser descodificado para que possa ser aplicado ao problema. Assim identificamos no cromossoma duas componentes, uma que representa a solução codificada e que é designada por *genótipo*, e outra que representa a solução descodificada e que é designada por *fenótipo* [5].

O material genético representado por um cromossoma é composto por *genes*. Os genes representam as características codificadas da solução do problema, isto é, na componente dos genótipos, enquanto que os *factores* representam as mesmas características na componente dos fenótipos [5]. A Figura 3.1 representa um cromossoma nas componentes dos fenótipos e dos genótipos, e a relação entre factores e genes.

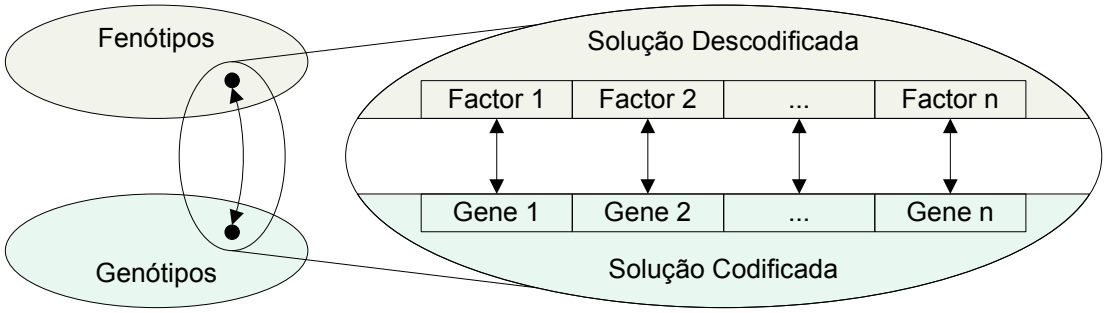

Figura 3.1: Representação de um cromossoma nas componentes dos fenótipos e dos genótipos.

Cada gene é composto por uma *string* de alelos, na qual a dimensão depende da codificação aplicada aos factores, sendo no mínimo de um alelo. A posição de um alelo nessa

string é designada por *locus*. Os *alelos* representam os valores possíveis pela codificação em uso, para cada locus num gene [5]. A Figura 3.2 representa um cromossoma com n genes, no qual cada gene contém vários alelos.

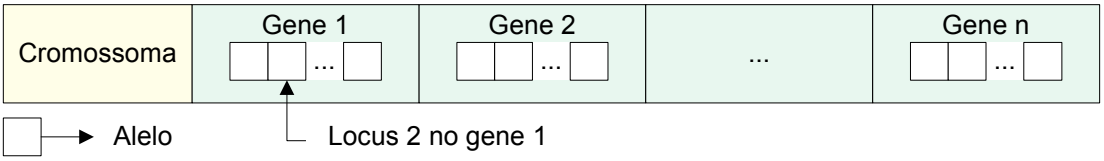

Figura 3.2: Representação de um cromossoma com genes e alelos, e o locus de um alelo.

Os *operadores genéticos* constituem um conjunto de métodos e regras que definem a manipulação do material genético dos cromossomas, para gerar novos cromossomas. Nas Secções 3.5 e 3.6, são apresentados os operadores genéticos de *cruzamento* e de *mutação*, respectivamente.

Segundo Mitchell [28], métodos robustos são aqueles que têm uma aplicabilidade quase universal. Estes métodos não necessitam de grandes alterações para serem utilizados numa grande variedade de problemas. Nesse sentido, os algoritmos genéticos são métodos robustos nos quais o nível de eficiência depende da codificação, dos operadores genéticos e da configuração dos diversos parâmetros específicos ao problema. No entanto, dada a sua robustez, um algoritmo genético pode funcionar com configurações muito distintas, mas é, geralmente, menos eficiente que os métodos construídos especificamente para um determinado problema [28].

A eficiência de um algoritmo genético pode ser melhorada em dois passos [31]. Num primeiro passo, são realizados pequenos ajustes aos processos de codificação, selecção, cruzamento, mutação e renovação de cromossomas na população, de forma a adequá-los ao problema. Por exemplo, a introdução de uma característica no processo de cruzamento, modificando-o e adaptando-o ao problema. Num segundo passo, temos o processo de afinação dos diversos parâmetros do algoritmo genético, tais como, o número de iterações, de elementos da população, de elementos a cruzar e a mutar, entre outros.

Apesar da eficiência de um algoritmo genético ser melhorada por estes dois passos, tornando-o mais eficiente para um determinado problema, este não deixa de ser robusto,

podendo ser aplicado a outros problemas.

A codificação usual para um cromossoma é a codificação binária. Este facto advém de Holland e outros investigadores terem usado esta codificação nos primeiros passos com algoritmos genéticos. As restantes secções deste capítulo vão considerar o uso da codificação binária para os cromossomas. No Capítulo 4 serão apresentadas as codificações por sequências de Prüfer e as codificações por sequências de arestas, ambas aplicadas a problemas com árvores de suporte.

3.1 Aptidão

Em qualquer problema de optimização necessitamos de métodos para testar e comparar as soluções. Os algoritmos genéticos não são excepção. No entanto existem diferentes designações para estes métodos.

Alguns autores [5, 9] designam estes métodos por funções de custo, enquanto outros [28, 37] designam por funções de aptidão. As funções de custo são, geralmente, associadas a problemas de minimização, isto é, pretendemos obter o cromossoma com o menor custo. As funções de aptidão são, geralmente, associadas a problemas de maximização, isto é, pretendemos obter o cromossoma mais apto. As funções de custo podem ser transformadas em funções de aptidão da mesma maneira como se transforma um problema de minimização num problema de maximização.

Existem ainda autores [28] que consideram apenas funções de aptidão, quer seja para problemas de minimização, quer seja para problemas de maximização. Nestes casos, a interpretação é a seguinte: *o cromossoma mais apto é aquele que minimiza os custos ou maximiza os lucros*. Portanto, a interpretação da aptidão depende do tipo de problema. Ao longo deste texto será considerada esta última interpretação para a aptidão de um cromossoma.

Para avaliar a aptidão de um cromossoma, é necessário descodificar o cromossoma, isto é, passar os valores da componente dos genótipos para a componente dos fenótipos [37]. Este cromossoma descodificado é avaliado por uma ou mais funções que vão determinar a sua aptidão. Estas funções calculam o custo/lucro e verificam se o custo/lucro e as respectivas soluções são admissíveis para o problema. Se a avaliação da aptidão determinar que o cromossoma é apto, isto significa que o cromossoma representa uma solução admissível

para o problema. Caso contrário, o cromossoma representa uma solução não admissível para o problema.

As funções de aptidão retornam valores numéricos. Isto permite estabelecer comparações entre os cromossomas para determinar qual o cromossoma mais apto [5].

A Figura 3.3 contém o diagrama de verificação de aptidão de um cromossoma, no qual o cromossoma é descodificado em factores, que são os argumentos da função de cálculo da aptidão. Posteriormente, com base no custo/lucro obtido e no cromossoma, é decidida a admissibilidade da solução.

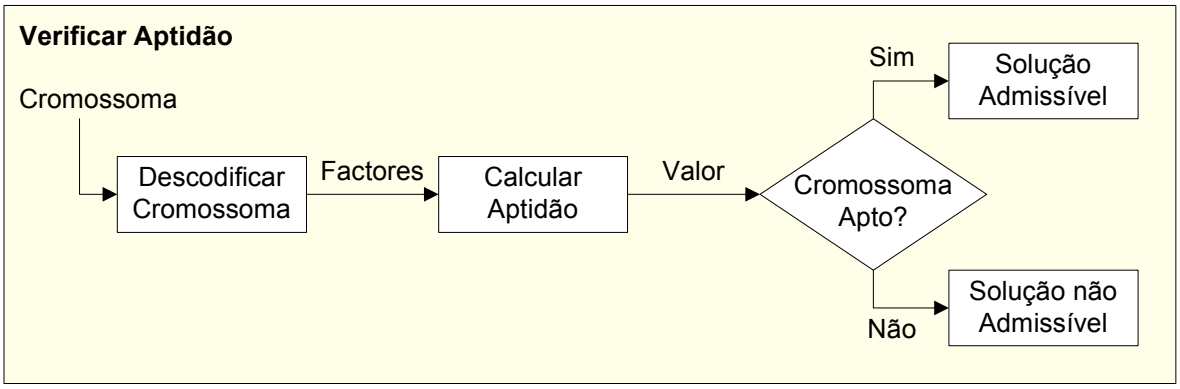

Figura 3.3: Diagrama de verificação da aptidão e admissibilidade de um cromossoma.

3.2 População

A população consiste num conjunto de cromossomas que representam soluções admissíveis para o problema [37]. Um elemento da população é, geralmente, designado por *indivíduo*.

Existem dois aspectos fundamentais sobre a população para os algoritmos genéticos [5]. O primeiro aspecto refere-se à dimensão da população. Enquanto que o segundo aspecto refere-se aos métodos para gerar a população inicial.

Ambos estes aspectos têm sido alvo de grande investigação. A dimensão da população depende de cada problema. No entanto se a dimensão da população for muito pequena, isto significa que existe pouca diversidade e o algoritmo genético vai convergir rapidamente para uma solução que pode representar um possível óptimo local, relativamente afastado do óptimo global. Se a dimensão da população for muito elevada, os custos computacionais

para gerar essa população ou para avaliar a sua aptidão, ou mesmo para o armazenamento dessa população, podem diminuir consideravelmente a velocidade de convergência do algoritmo genético, tornando-o inaceitável. Estes custos computacionais, ao nível do tempo e da memória, têm de ser cuidadosamente equilibrados. Poli et al. [31] referem que, empiricamente, a dimensão da população deve ser a maior possível, com a carga computacional aceitável e que é frequente a dimensão da população ser de 500 indivíduos. No entanto, referem que populações muito menores também são comuns, mas que estas recorrem mais a operadores genéticos de mutação, comparativamente aos operadores genéticos de cruzamento. Mitchell [28] cita vários estudos, por exemplo, De Jong (1975) propunha uma população ideal entre 50 a 100 indivíduos; um outro exemplo, Grefenstette (1986) e Bramlette (1991) propunham uma população ideal com 30 indivíduos; por fim apenas Goldberg (1989) propunha populações com dimensões maiores. Note-se que os livros de Poli et al. [31] e de Mitchell [28] estão separados por mais de uma década. O primeiro livro [31], mais recente, cita dimensões para a população bem superiores às dimensões citadas no segundo livro [28]. Uma das razões para tal deve-se ao aumento do poder computacional das ferramentas usadas nos algoritmos genéticos durante esse período de mais de uma década.

A geração da população é geralmente aleatória. No entanto existem problemas onde a geração aleatória da população é pouco eficaz, sendo vantajoso recorrer a outros métodos como é o caso das *heurísticas* [5, 28, 31, 37]. As heurísticas podem gerar mais rapidamente a população, ou gerar uma população na qual a média de aptidão dos indivíduos seja superior à de uma geração aleatória.

Existem, ainda, algoritmos genéticos mais sofisticados nos quais a dimensão da população varia consoante a diversidade dos indivíduos e taxa de convergência [31].

3.3 Evolução

O passo seguinte à geração da população, consiste em fazer evoluir a população através de operadores genéticos, gerando novos indivíduos e substituindo os indivíduos menos aptos [5]. Os indivíduos gerados e admissíveis pertencem ao *espaço de pesquisa* do problema. O espaço de pesquisa corresponde à região de admissibilidade do problema, isto é, ao espaço onde se encontram todas as soluções admissíveis e, de entre as quais, uma ou mais soluções óptimas para o problema [31].

Um algoritmo genético deve ser capaz de explorar este espaço o máximo possível. O processo que permite explorar o espaço de pesquisa é considerado o núcleo ou o motor do algoritmo genético [31].

O processo que simula, num algoritmo genético, a selecção natural que ocorre na natureza, é composto por quatro passos [37] (o autor não cita o passo da mutação):

1. Selecção - Seleccionar um conjunto de pais.
2. Cruzamento - Cruzar os pais para obter filhos.
3. Mutação - Eventualmente aplicar uma mutação aos filhos.
4. Renovação - Substituir alguns indivíduos da população por estes filhos.

Estes quatro passos constituem o chamado *ciclo de procriação*. Cada ciclo de procriação é responsável por uma nova geração de indivíduos para a população, que substituem os indivíduos menos aptos nela existentes. Note-se que o passo da mutação não é aplicado em todas as iterações do algoritmo genético, isto é, a mutação é um passo baseado num parâmetro percentagem de mutação na população (Secção 3.6).

Podemos associar a cada iteração do algoritmo genético um ciclo de procriação e por extensão uma geração. É frequente na literatura, este número de iterações do algoritmo genético ser designado por número de gerações. Além disso, determinar o número de gerações a usar num algoritmo genético tem tanta importância como a dimensão da população. Ambos estes parâmetros têm influência na convergência do algoritmo genético. Se o número de gerações for muito baixo, não conseguimos explorar todo o espaço de pesquisa. Por outro lado, se o número de gerações for muito alto, podemos explorar mais o espaço de pesquisa, mas os custos computacionais podem ser demasiado elevados [31].

3.4 Selecção

A selecção é o primeiro passo do ciclo de procriação. Consiste em seleccionar indivíduos da população que serão recombinados para dar origem a novos indivíduos. A recombinação de indivíduos da população constitui o segundo passo do ciclo de procriação e será descrita na Secção 3.5. O processo de selecção pretende simular a selecção natural que ocorre na natureza, onde os indivíduos mais aptos são os que sobrevivem.

Poderíamos pensar que bastaria seleccionar os indivíduos mais aptos para o processo de cruzamento. Mas na natureza alguns indivíduos menos aptos também são seleccionados, quer seja pelo factor sorte ou por outros factores circunstanciais [37]. Assim o processo de selecção não é determinista, isto é, não vamos seleccionar apenas os indivíduos mais aptos. É um processo aleatório que selecciona indivíduos da população com base na sua aptidão e que tende a favorecer os indivíduos mais aptos como na natureza.

Existem vários métodos para o processo de selecção, sendo dois dos mais usados o *Método da Roleta* (*Roulette Wheel Selection*) e o *Método do Torneio* (*Tournament Selection*).

Num algoritmo genético não é comum que todos os indivíduos da população sejam seleccionados para posterior cruzamento [28]. Assim, parte da nova população é constituída por indivíduos da população actual e outra parte por indivíduos obtidos por cruzamento e/ou mutação. A percentagem de novos indivíduos é chamada de *amplitude geracional* (*generation gap*) [28] e depende do número de operações de cruzamento e/ou mutação efectuadas, assim como do método de renovação da população utilizado.

O *Método da Roleta* é apresentado na Secção 3.4.1, enquanto que o *Método do Torneio* é apresentado na Secção 3.4.2. Em [28] podem ser consultados outros métodos de selecção de indivíduos da população para as operações genéticas de cruzamento e/ou mutação.

3.4.1 Método da Roleta

No método de selecção da Roleta (*Roulette Wheel Selection*) a probabilidade de selecção de um indivíduo da população é proporcional à sua aptidão. O método de selecção da Roleta consiste em [37]:

1. determinar o somatório das aptidões dos indivíduos da população, S;
2. gerar um número aleatório, R uniformemente distribuído entre 0 e S;
3. adicionar sucessivamente as aptidões dos indivíduos da população, até que o somatório seja superior a R. O último indivíduo adicionado é o indivíduo seleccionado para o grupo de indivíduos a usar na operação de cruzamento.

Os pontos 2 e 3 repetem-se até termos um grupo completo de indivíduos para a operação de cruzamento.

A analogia à roleta consiste em distribuir a proporção associada a cada indivíduo da população sobre um disco giratório. A amplitude de cada casa da roleta é dada pela proporção associada a cada indivíduo da população. Cada jogo representa a geração de um número aleatório com distribuição uniforme. Esse número aleatório está associado à proporção do indivíduo na população, permitindo que o mesmo seja seleccionado.

Sivanandam e Deepa [37] indicam que não é um método adequado para populações de grandes dimensões, devido a ser necessário percorrer todos os indivíduos para calcular o somatório das aptidões da população.

Exemplo 3.1.

Aplicação do método de selecção da Roleta ao problema de maximização da função $f(x) = -x^2 + 16x, x \in [0, 15]$*, seleccionando 2 indivíduos, e considerando uma população com 6 indivíduos e codificação binária de 4 bits.*

Passo 1: Obter o somatório das aptidões dos indivíduos da população, S, representado na Tabela 3.1 (considerando que a função de aptidão é $f(x)$).

	Cromossoma	Valor Decimal, x	Aptidão, $f(x)$	Percentagem na Roleta, %
A	1011	11	55	20.22
B	0011	3	39	14.34
C	1000	8	64	23.53
D	0110	6	60	22.06
E	0001	1	15	5.51
F	1101	13	39	14.34
Total	–	–	$S = 272$	100.00

Tabela 3.1: Representação da aptidão e probabilidade de selecção de um indivíduo da população, usando o método de selecção da Roleta.

Passo 2: Gerar dois números aleatórios, R_1 e R_2, com distribuição uniforme entre 0 e S. Cada número aleatório corresponde a um indivíduo da população a seleccionar. Consideremos $R_1 = 82$ e $R_2 = 173$.

Passo 3: Adicionar as aptidões dos indivíduos da população, até que o somatório seja superior a R.

Os cromossomas seleccionados pelo método de selecção da Roleta são o B e o D, como representado na Figura 3.4.

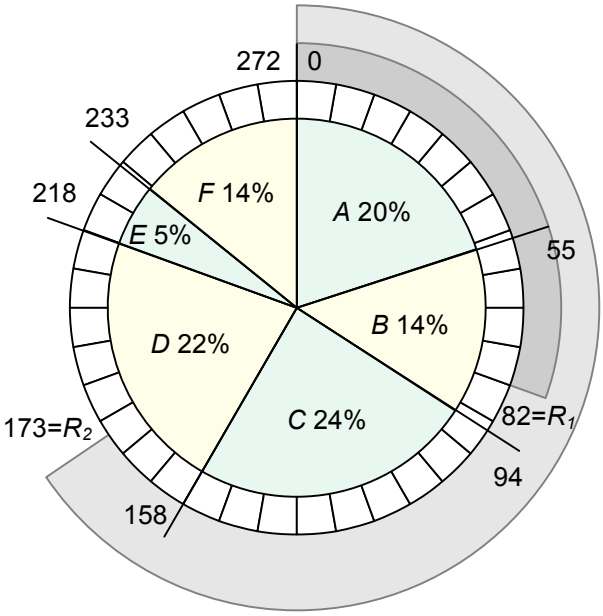

Figura 3.4: Representação do método de selecção da Roleta para o Exemplo 3.1.

□

Note-se que a população, para este método de selecção, não necessita de estar ordenada por aptidão. A Figura 3.4 apresenta um exemplo do método de selecção da Roleta, onde o número aleatório a gerar, R, pode tomar valores entre 0 e 272. O indivíduo B tem uma aptidão com valor 39, cuja soma sucessiva ao indivíduo A, coloca-o com valores de selecção entre 55 e 94. Como R tem distribuição uniforme entre 0 e 272, a probabilidade de selecção do indivíduo B corresponde à amplitude $94 - 55 = 39$ a dividir por 272, sendo de 14%.

Cada jogo representa a selecção de um indivíduo da população. O número de jogos realizados é denotado por *pressão de selecção* [2]. A pressão de selecção está associada ao nível de favorecimento dos indivíduos mais aptos. Quando a pressão de selecção é elevada, serão seleccionados mais indivíduos cuja aptidão é elevada. No entanto se for demasiado alta, o algoritmo genético pode convergir rapidamente para uma solução localmente óptima. Se for muito baixa, o algoritmo genético poderá levar demasiado tempo a convergir para a solução.

3.4.2 Método do Torneio

O método de selecção do Torneio (*Tournament Selection*) consiste em seleccionar indivíduos da população, calcular as suas aptidões e escolher o mais apto. O método de selecção do Torneio consiste em [12]:

1. seleccionar aleatoriamente dois ou mais indivíduos da população;
2. calcular as aptidões dos indivíduos seleccionados;
3. indicar o indivíduo mais apto que é seleccionado para o grupo de indivíduos a usar na operação de cruzamento.

O torneio repete-se até termos um grupo completo de indivíduos para a operação de cruzamento.

A *dimensão da selecção* consiste no número de indivíduos escolhidos para a realização do torneio, onde apenas o mais apto é seleccionado [12]. Se a dimensão da selecção for elevada, existe uma maior probabilidade de serem escolhidos indivíduos cuja aptidão seja elevada. Repetido este tipo de selecção de indivíduos de elevada aptidão durante várias gerações, previne o algoritmo genético de explorar todo o espaço de pesquisa, centrando-se num sub-espaço constituído por estes indivíduos com aptidões elevadas e cada vez mais semelhantes. Neste caso o algoritmo genético tende a convergir rapidamente para um possível óptimo local. Se a dimensão da selecção for baixa, existe uma maior probabilidade de serem escolhidos indivíduos cuja aptidão seja baixa. Neste caso o algoritmo genético poderá levar demasiado tempo a convergir para a solução.

Goldberg e Deb [12] estudaram o número de indivíduos a seleccionar aleatoriamente em cada torneio (dimensão da selecção) e concluíram que quanto maior for esse número, menor será o tempo necessário para o algoritmo genético convergir. No entanto esses ganhos são logarítmicos, sendo frequente a dimensão do torneio ser de dois ou três indivíduos.

Exemplo 3.2.

Aplicação o método de selecção do Torneio ao problema de maximização da função $f(x) = -x^2 + 16x, x \in [0, 15]$*, seleccionando 2 indivíduos, e considerando uma população com 6 indivíduos e codificação binária de 4 bits.*

Passo 1: Seleccionar, aleatoriamente, dois indivíduos da população, em dois torneios, como representado na Figura 3.5.

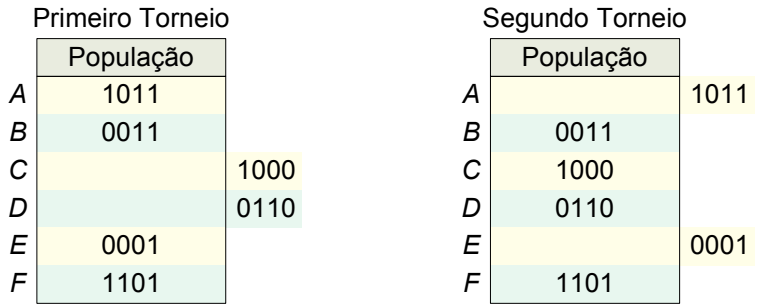

Figura 3.5: Representação de dois torneios para o método de selecção do Torneio.

Os indivíduos seleccionados no primeiro torneio são o C e o D, e no segundo torneio são o A e o E.

Passo 2: Calcular as aptidões dos indivíduos seleccionados em cada torneio (considerando que a função de aptidão é $f(x)$).
Com base no Exemplo 3.1, as aptidões dos indivíduos seleccionados no primeiro torneio, C e D, são 64 e 60, e as aptidões dos indivíduos seleccionados no segundo torneio, A e E, são 55 e 15.

Passo 3: Indicar o indivíduo mais apto de cada torneio.
O indivíduo com maior aptidão no primeiro torneio é o C, e no segundo torneio é o A, sendo estes os cromossomas seleccionados pelo método de selecção do Torneio.

□

3.5 Cruzamento

O cruzamento é o segundo passo do ciclo de procriação. Consiste em seleccionar aleatoriamente um par de pais do grupo de indivíduos seleccionados no primeiro passo do ciclo de procriação. De seguida é aplicado um método que permite misturar o material genético de ambos os pais para obter os filhos.

Nas próximas subsecções serão descritos alguns dos métodos de cruzamento mais conhecidos e utilizados, tais como o *One Point Crossover*, o *Two Point Crossover*, o *K-Point*

Crossover e o *Uniform Crossover*. Estes métodos caracterizam-se por poderem ser aplicados a várias codificações [28]. Na Secção 4.3 são referidos os métodos de cruzamento *PrimRST*, *KruskalRST* e *RandWalkRST*, que são específicos para a codificação de cromossomas por sequências de arestas, também apresentada na mesma secção. Em [37] podem ser consultados outros métodos de cruzamento de cromossomas.

Vamos considerar que um cromossoma tem uma *string* binária com comprimento de n bits. Assim o locus varia entre 1, para a primeira posição na *string*, e n, para a última posição na *string*.

3.5.1 One Point Crossover

O método de cruzamento *One Point Crossover* [5] consiste em seleccionar um *ponto de corte* aleatório entre os locus 1 e n, para os cromossomas pai e mãe.

O material genético do primeiro filho é composto pelo material genético do pai, à esquerda do ponto de corte, e pelo material genético da mãe, à direita do ponto de corte. O material genético do segundo filho é composto pelo material genético do pai, à direita do ponto de corte, e pelo material genético da mãe, à esquerda do ponto de corte, como representado na Figura 3.6.

Embora seja um dos métodos mais usados e simples de implementar, Mitchell [28] indica um problema associado. Como o método selecciona um ponto de corte entre 1 e n, então os valores em 1 e em n vão ser propagados pelos filhos em diversas gerações. Como consequência, uma parte do espaço de pesquisa é ignorada. Essa parte do espaço de pesquisa, só poderá ser acedida pela mutação dos filhos.

3.5.2 Two Point Crossover

O método de cruzamento *Two Point Crossover* [5] tenta evitar o problema do *One Point Crossover*. Neste método são seleccionados aleatoriamente dois pontos de corte. O primeiro filho é composto pelo material genético do pai, exterior aos dois pontos de corte, e pelo material genético da mãe, interior aos dois pontos de corte. O segundo filho é composto pelo material genético do pai, interior aos dois pontos de corte, e pelo material genético da mãe, exterior aos dois pontos de corte, como representado na Figura 3.6.

Este método tende a abrandar a convergência do algoritmo genético, porque ao retirar e adicionar partes de um cromossoma, a aptidão elevada de um dos pais pode ser comprometida pela parte fraca adicionada pelo outro progenitor. No entanto esta desvantagem temporal do algoritmo genético, é compensada pela maior exploração do espaço de pesquisa, podendo o algoritmo genético convergir mais perto da solução óptima.

3.5.3 K-Point Crossover

O método de cruzamento *K-Point Crossover* [5] é uma generalização a K pontos de corte dos métodos *One Point Crossover* e *Two Point Crossover*. Assim ambos os filhos serão compostos por pequenos segmentos alternados de material genético de cada um dos pais, como representado na Figura 3.6.

De forma semelhante ao método *Two Point Crossover*, a convergência do algoritmo genético abranda, tendo o benefício de explorar melhor o espaço de pesquisa. No entanto este método é computacionalmente mais moroso que o método *Two Point Crossover*.

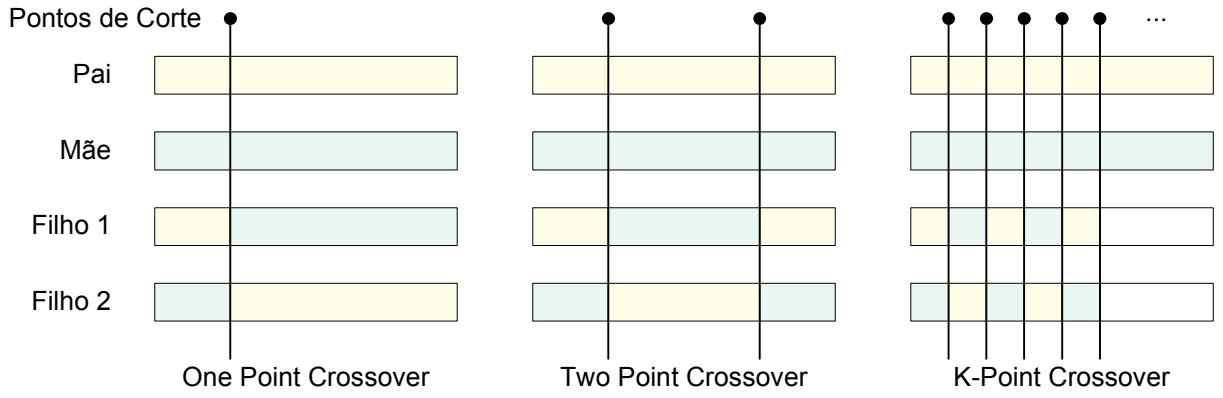

Figura 3.6: Representação dos métodos de cruzamento *One Point Crossover*, *Two Point Crossover* e *K-Point Crossover*.

3.5.4 Uniform Crossover

O método de cruzamento *Uniform Crossover* [37] utiliza uma máscara binária para o processo de cruzamento. A máscara tem a mesma dimensão que as *strings* dos cromossomas pai e mãe. Ao pai é associado o valor 0 e à mãe é associado o valor 1. A máscara é gerada aleatoriamente com estes valores. Os filhos são construídos percorrendo todas as posições da máscara.

Assim se na posição actual da máscara temos um valor 0, o primeiro filho recebe, na mesma posição, o alelo do pai e o segundo filho recebe o alelo da mãe. Se na posição actual da máscara temos um valor 1, o primeiro filho recebe, na mesma posição, o alelo da mãe e o segundo filho recebe o alelo do pai, como representado na Figura 3.7.

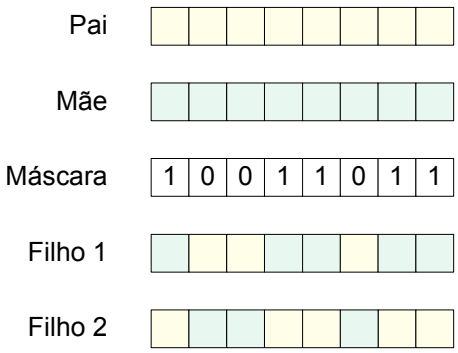

Figura 3.7: Representação do método de cruzamento *Uniform Crossover*.

Sivanandam e Deepa [37] indicam que quando a máscara é renovada para cada operação de cruzamento, a mistura do material genético nos filhos tende a ser metade de cada progenitor.

3.6 Mutação

A mutação é o terceiro passo do ciclo de procriação. A mutação consiste em mudar, aleatoriamente, uma parte do material genético de um cromossoma. O método de mutação *Single Point Mutation* [5] consiste no seguinte. É gerada, aleatoriamente, uma posição no cromossoma. Nessa posição é gerado, novamente, um valor aleatório para substituir o alelo actual. Este novo alelo tem de respeitar a codificação actual do cromossoma. Numa codificação binária, um alelo tem apenas dois valores, 0 e 1.

Como referido em secções anteriores, a pressão de selecção (para o método da Roleta) e a dimensão da selecção (para o método do Torneio), tendem a influenciar a velocidade de convergência do algoritmo genético. Se os métodos de selecção seleccionam repetidamente indivíduos com aptidões elevadas, após algumas gerações os restantes indivíduos da população tendem a aproximar-se desses indivíduos com aptidões elevadas. Por um lado o espaço de pesquisa deixa de ser totalmente explorado e por outro lado os indivíduos

da população tornam-se iguais implicando a convergência do algoritmo genético para um possível óptimo local [37].

A mutação é um mecanismo que permite diversificar a componente genética da população, impedindo que o algoritmo genético convirja rapidamente para soluções localmente óptimas.

A mutação deve ser ajustada a cada problema, tendo, geralmente, dois parâmetros associados [37]. Um parâmetro para a percentagem de mutação do cromossoma e outro parâmetro para a percentagem de mutação na população. O primeiro parâmetro indica, em média, quantos alelos vão ser mutados num único cromossoma. O segundo parâmetro indica, em média, quantos cromossomas da população recebem uma mutação. Se o valor destes parâmetros for muito alto, o algoritmo genético comporta-se como um algoritmo de pesquisa aleatória (*Random Search Algorithm*) [37]. Por isso é frequente alterar apenas um alelo num cromossoma, enquanto que o segundo parâmetro varia de problema para problema. Em problemas onde a população tende a convergir rapidamente para uma solução localmente óptima, a percentagem de mutação deve ser maior para permitir uma melhor exploração da região de admissibilidade desses problemas.

Na Figura 3.8 está representado um cromossoma com codificação binária de 8 bits, mutado no locus 6.

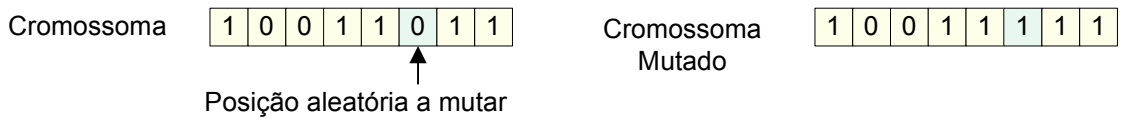

Figura 3.8: Exemplo da mutação *Single Point Mutation* num cromossoma com codificação binária.

3.7 Renovação

A renovação é o quarto passo do ciclo de procriação. Consiste em introduzir na população actual os novos indivíduos obtidos por cruzamento e/ou mutação, após a avaliação da sua aptidão. A renovação pode ser realizada individualmente ou em bloco [31].

A renovação individual consiste em substituir um indivíduo da população imediatamente após um novo indivíduo ter sido gerado por cruzamento e/ou mutação. Por outro lado a renovação em bloco consiste em substituir os indivíduos da população após todos os novos indivíduos terem sido gerados por cruzamento e/ou mutação na iteração actual.

Existem vários métodos para a substituição dos indivíduos da população [37]:

1. O primeiro método consiste em substituir aleatoriamente os indivíduos da população. Este método é útil em populações de pequena dimensão pois pode introduzir indivíduos menos aptos, contribuindo para a diversidade genética da população.
2. O segundo método consiste na substituição de pais com aptidão baixa. Considerando que um par de pais pode gerar um par de filhos, estes filhos substituem os pais se as suas aptidões forem superiores.
3. O terceiro método consiste em substituir sempre os pais pelos filhos, independentemente das suas aptidões. Este método deve ser associado a métodos de selecção que não favoreçam fortemente os indivíduos mais aptos, caso contrário, o algoritmo converge prematuramente para um possível óptimo local.
4. Um quarto método consiste em substituir sempre os indivíduos menos aptos. Este método tende a favorecer os indivíduos mais aptos.

3.8 Elitismo

Os métodos de selecção da Roleta e do Torneio, não garantem a selecção do indivíduo mais apto da população. Dependendo do tipo de renovação da população, podemos acabar por perder o indivíduo mais apto da geração actual [5]. Como consequência o algoritmo genético vai convergir mais lentamente. Em alguns problemas a convergência mais lenta permite explorar melhor o espaço de pesquisa. Noutros problemas existem vantagens em não perder o indivíduo mais apto da geração actual. Este indivíduo mais apto da geração actual é designado por indivíduo *elite*.

O indivíduo elite deve ser propagado para a nova geração, caso ainda seja o indivíduo mais apto. Este processo tem o nome de *elitismo* [5].

3.9 Convergência

Um algoritmo genético converge, quando a aptidão dos indivíduos da população deixa de melhorar através do processo de cruzamento e apenas melhora através do processo de mutação. Nesta situação a aptidão dos indivíduos através do cruzamento mantém-se. Qualquer cruzamento de indivíduos gera um novo indivíduo com a mesma aptidão, logo, só é possível obter novos indivíduos para a população através da mutação. Quando tal acontece o algoritmo genético deve parar.

Os critérios típicos de paragem dos algoritmos genéticos são os seguintes [37]:

1. O número máximo de gerações foi atingido.
2. O tempo máximo de processamento foi excedido.
3. A média da aptidão da população manteve-se inalterada durante um certo número de gerações.
4. A média da aptidão da população manteve-se inalterada durante um certo limite de tempo.

Após a paragem do algoritmo genético, estes tipicamente apresentam [31]:

1. O indivíduo mais apto ou um conjunto com os indivíduos mais aptos e as respectivas aptidões.
2. O indivíduo menos apto ou um conjunto com os indivíduos menos aptos e as respectivas aptidões.
3. O somatório das aptidões dos indivíduos da população.
4. A média das aptidões dos indivíduos da população.

Capítulo 4

Codificações

A *codificação* consiste num conjunto de regras ou algoritmos que permitem passar da representação de um cromossoma na componente dos fenótipos, para a representação de um cromossoma na componente dos genótipos. O processo inverso é denominado de *descodificação*.

No contexto dos algoritmos genéticos, a codificação permite representar dados tratados manualmente, em dados interpretáveis por um computador.

As codificações com números binários ou, simplesmente, codificações binárias são as que mais se aproximam da linguagem máquina dos computadores. Historicamente, foram das primeiras codificações a serem usadas nos algoritmos genéticos, inclusive nos primeiros trabalhos sobre algoritmos genéticos, de John Holland e de David Goldberg, referidos no Capítulo 3.

Posteriormente, com o aumento do poder de processamento dos computadores e introdução de novas linguagens de programação, foram surgindo outros tipos de codificações de cromossomas que melhor se adaptam a cada problema. Na Secção 1.2 foram mencionadas algumas referências bibliográficas de problemas de árvores de suporte com restrições resolvidos recorrendo a algoritmos genéticos. Nesses problemas são aplicadas a codificação por sequências de Prüfer [26], a codificação por sequências de arestas [34], a codificação por permutações [22] e a codificação por predecessores [33] que apresentam um ponto comum, nomeadamente são codificações por sequências de números inteiros. Em [23] são usados cromossomas com codificação por pesos, que correspondem a sequências de números reais. Na literatura encontramos outros tipos de codificações, com estruturas mais complexas que se adaptam aos respectivos problemas. Uma dessas codificações é a codificação em árvore,

ou árvore de sintaxe, aplicada a problemas de inferência gramatical e reconhecimento de padrões.

Neste capítulo são apresentadas as propriedades das codificações na Secção 4.1, a codificação por sequências de Prüfer na Secção 4.2 e a codificação por sequências de arestas na Secção 4.3. Ambas as codificações são aplicadas a problemas de determinação da *Árvore de Suporte de Custo Mínimo*, como referido no capítulo introdutório. No Capítulo 5, estas codificações serão aplicadas ao problema da determinação da *Árvore de Suporte de Custo Mínimo com Restrições de Salto*.

A título introdutório e no seguimento do capítulo anterior, é exemplificada a codificação binária nos Exemplos 4.1 e 4.2.

O Exemplo 4.1 indica como corresponder um conjunto de valores inteiros para um conjunto de valores binários.

Exemplo 4.1.

Aplicação da codificação binária à variável x, representando-a por uma string *de bits, sabendo que $x \in \{0, 1, \ldots, 15\}$.*

O maior número inteiro que podemos codificar com n bits é dado por $N_{max} = 2^n - 1$. Logo para codificar N_{max} necessitamos de $n = Log_2(N_{max} + 1)$ bits. Assim se $N_{max} = 15$, temos $n = Log_2(16) = 4$ bits. A correspondência entre o valor inteiro e o valor binário é representada na Tabela 4.1.

Valor Binário $\langle b_3 b_2 b_1 b_0 \rangle_2$	Valor Inteiro x	Valor Binário $\langle b_3 b_2 b_1 b_0 \rangle_2$	Valor Inteiro x
0000	0	1000	8
0001	1	1001	9
0010	2	1010	10
0011	3	1011	11
0100	4	1100	12
0101	5	1101	13
0110	6	1110	14
0111	7	1111	15

Tabela 4.1: Representação dos cromossomas com codificação binária de 4 bits de valores inteiros.

□

No Exemplo 4.1 o conjunto de valores que a variável x pode assumir corresponde ao número de valores possíveis de codificar com 4 bits. No entanto, muitas das vezes, estamos a trabalhar com valores em $\mathbb{R}$. A correspondência entre valores binários e valores reais realiza-se em dois passos [5]:

1. converter a *string* binária $\langle b_n \dots b_1 b_0 \rangle_2$ para um número inteiro x',

$$\langle b_n \dots b_1 b_0 \rangle_2 = \left(\sum_{i=0}^{n} b_i 2^i \right)_{10} = x' \tag{4.1}$$

2. corresponder a cada número inteiro x' um número real $x \in [a, b]$, onde a e b são os limites do intervalo de valores possíveis para x,

$$x = \text{arredondar} \left\{ a + \frac{x' (b - a)}{2^n - 1} \right\} \tag{4.2}$$

O Exemplo 4.2 indica como corresponder um conjunto de valores reais a um conjunto de valores binários.

Exemplo 4.2.

Aplicação a codificação binária à variável x, representando-a por uma string *de 4 bits, sabendo que $x \in \{0, 0.1, \dots, 1.0\}$.*

A variável x pode tomar 11 valores reais e a codificação binária com 4 bits permite codificar 16 valores. Neste caso, a correspondência entre os valores binários e valores reais não será de um-para-um, como no Exemplo 4.1. A correspondência entre o valor binário e o valor real, realiza-se em dois passos, usando as Equações (4.1) e (4.2), e é representada na Tabela 4.2, com $a = 0$ e $b = 1.0$.

□

No Exemplo 4.2 verificamos que alguns valores reais são codificados por dois valores binários. Por exemplo, o valor real 0.1 é codificado pelos valores binários 0001 e 0010. Esta situação aparentemente inofensiva, levanta alguns problemas para os algoritmos genéticos, que serão expostos na Secção 4.1, juntamente com a descrição das propriedades para as codificações. Note-se, ainda, que os Exemplos 4.1 e 4.2 são baseados nas Secções 1.4 e 2.3 de [9].

Valor Binário $\langle b_3 b_2 b_1 b_0 \rangle_2$	Valor Inteiro x'	Valor Real x	Valor Binário $\langle b_3 b_2 b_1 b_0 \rangle_2$	Valor Inteiro x'	Valor Real x
0000	0	0	1000	8	0.5
0001	1	0.1	1001	9	0.6
0010	2	0.1	1010	10	0.7
0011	3	0.2	1011	11	0.7
0100	4	0.3	1100	12	0.8
0101	5	0.3	1101	13	0.9
0110	6	0.4	1110	14	0.9
0111	7	0.5	1111	15	1.0

Tabela 4.2: Representação dos cromossomas com codificação binária de 4 bits de valores reais.

4.1 Propriedades das Codificações

Os autores Ingo Rechenberg e Satoh Kobayashi apresentaram as propriedades que devem ser respeitadas pelas codificações de cromossomas. Estas foram posteriormente compiladas em [9, 15, 30], e consistem nas seguintes propriedades:

- *eficiência* (*efficiency*);
- *completude* (*completeness*);
- *não redundância* (*non-redundancy*);
- *legalidade* (*legality*);
- *causalidade* (*causality*);
- *hereditariedade* (*heritability*).

A propriedade da *eficiência* [15] indica que os algoritmos de codificação e descodificação dos indivíduos da população devem ser eficientes, de forma a não prejudicar a performance global do algoritmo genético.

A propriedade da *completude* [9, 30] indica que qualquer solução do espaço de pesquisa tem uma solução codificada, garantindo por parte do algoritmo genético, a acessibilidade a todos os indivíduos da região de admissibilidade.

Num algoritmo genético podemos estabelecer três tipos de correspondência entre indivíduos da componente dos fenótipos e indivíduos da componente dos genótipos, nomeadamente a correspondência *muitos-para-um*, *um-para-muitos* e *um-para-um*.

A correspondência *muitos-para-um* indica que um elemento da componente dos genótipos codifica vários elementos da componente dos fenótipos. Neste caso quando o algoritmo genético converge, o indivíduo com a melhor aptidão corresponde a várias soluções para a função objectivo. O algoritmo genético necessita pesquisar qual dessas soluções corresponde à melhor solução no espaço dos fenótipos.

A correspondência *um-para-muitos* indica que um elemento da componente dos fenótipos é codificado por vários elementos da componente dos genótipos. Neste caso o algoritmo genético perde tempo ao efectuar a pesquisa sobre vários indivíduos que correspondem a uma mesma solução.

A correspondência *um-para-um* indica que cada elemento da componente dos fenótipos é codificado por um único elemento da componente dos genótipos. Esta correspondência unívoca entre um elemento de cada componente corresponde à propriedade da *não redundância* [9]. Note-se que a codificação empregue no Exemplo 4.2 falha a propriedade da *não redundância*.

A propriedade da *legalidade* [9] indica que qualquer permutação num indivíduo codificado, é ainda uma solução. Esta propriedade é de grande importância para os operadores genéticos de cruzamento e mutação, os quais permutam e combinam indivíduos codificados para gerar novos indivíduos codificados.

A propriedade da *causalidade* [9], também conhecida como *localidade* (*locality*) [15, 30], está ligada às variações propagadas entre a componente dos fenótipos e a componente dos genótipos. Isto significa que se forem alterados n genes de um cromossoma na componente dos genótipos, os factores correspondentes a esses n genes na componente dos fenótipos também são alterados, analogamente no sentido da componente dos fenótipos para a componente dos genótipos. Neste caso diz-se que existe *causalidade forte*, sendo a preferencial. Se forem alterados n genes/factores de um cromossoma e não se verificarem alterações em todos os n factores/genes correspondentes, então diz-se que existe *causalidade fraca*. Se forem alterados n genes/factores de um cromossoma e não se verificarem alterações nos n factores/genes correspondentes, então não existe causalidade.

A propriedade da *hereditariedade* [15], também conhecida como propriedade *Lamarckiana* (*Lamarckian property*) [9], indica que os genes que constituem um indivíduo codificado

(cromossoma) têm significado independente do contexto. Por outras palavras, esta propriedade procura que uma determinada característica de um pai seja passada para o seu filho, através da operação de cruzamento, sem que seja perdido o seu significado.

Na prática existem codificações que não respeitam estas propriedades. Das duas codificações em estudo (sequências de Prüfer e sequências de arestas) e considerando o problema da árvore de suporte sem restrições, apenas a codificação por sequências de Prüfer não cumpre na totalidade estas propriedades, pois não respeita nem a propriedade da *causalidade*, nem a propriedade da *hereditariedade*, como veremos na Secção 4.2.

4.2 Codificação por Sequências de Prüfer

As sequências ou códigos de Prüfer surgiram em 1918 e foram desenvolvidas por Ernst Heinz Prüfer numa das várias demonstrações existentes para a *Fórmula de Cayley*.

A *Fórmula de Cayley* foi publicada em 1889 por Arthur Cayley, é uma versão revista do teorema de Carl Wilhelm Borchardt de 1860, considerando os graus dos vértices e enunciada de seguida, cuja prova pode ser consultada em [4].

Teorema 4.1 (Fórmula de Cayley)**.**

Seja K_n um grafo completo de ordem $n \in \mathbb{N}$. O número de árvores de suporte para o grafo K_n é n^{n-2} e denota-se por $\tau(K_n) = n^{n-2}$.

Para codificar uma árvore de suporte, com n vértices, numa sequência de Prüfer, consideramos a árvore etiquetada com os vértices de 1 até n e seguimos os passos apresentados no Algoritmo 4.1 [4]. Note-se que o Algoritmo 4.1 usa alguns conceitos apresentados no Capítulo 2, nomeadamente os conceitos de árvore de suporte (Definição 2.10), ordem de um grafo (Definição 2.2), grau de um vértice (Definição 2.3) e vizinhança de um vértice (Definição 2.4).

O processo, apresentado no Algoritmo 4.1, consiste em cada iteração eliminar o vértice com a etiqueta de menor valor, cujo grau desse vértice não exceda um e colocar a etiqueta do seu vizinho na sequência de Prüfer, até que o número de vértices na árvore seja apenas dois. Este processo é exemplificado no Exemplo 4.3.

Algoritmo 4.1: Codificar uma árvore de suporte numa sequência de Prüfer.

Entrada: $T = (V, E) \rightarrow$ árvore de suporte
Saída: $p \rightarrow$ conjunto com a sequência de Prüfer
Variáveis: $n \rightarrow$ número de vértices na árvore de suporte T
$R \rightarrow$ árvore de suporte temporária
$i \rightarrow$ número de iteração actual
$s, u \rightarrow$ vértices temporários

1 $p \leftarrow \{\}$
2 $n \leftarrow |V(T)|$
3 $R \leftarrow T$
4 **para** $i \leftarrow 1$ **até** $n - 2$ **fazer**
5 $\quad s \leftarrow \min\{u \in V(R) : d_R(u) = 1\}$
6 $\quad p[i] \leftarrow N_R(s)$
7 $\quad R \leftarrow R - s$ `//remover o vértice` s `e a aresta incidente da árvore` R
8 **devolver** p

Exemplo 4.3.

Codificar a árvore de suporte representada na Figura 4.1 numa sequência de Prüfer, recorrendo ao Algoritmo 4.1.

Na Tabela 4.3 são apresentados os passos descritos no Algoritmo 4.1 enquanto que a Figura 4.2 representa graficamente os passos executados na Tabela 4.3.

Na Figura 4.2 estão representados o vértice a eliminar e o vértice vizinho em cada iteração. A vermelho está representado o vértice a eliminar, juntamente com a aresta que lhe é incidente. A verde está representado o vértice vizinho, que entra para a sequência de Prüfer.

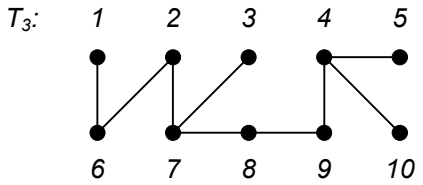

Figura 4.1: Representação da árvore de suporte do Exemplo 4.3.

A sequência de Prüfer para a árvore de suporte T_3 representada na Figura 4.1 é dada por $\{6, 7, 4, 2, 7, 8, 9, 4\}$.

□

Início	$p \leftarrow \{\}\,,\ n \leftarrow \lvert V(T)\rvert = 10,\ R \leftarrow T$
Iteração 1	$i \leftarrow 1,\ s \leftarrow \min\{1,3,5,10\} = 1,\ p[1] \leftarrow N_R(1) = 6,\ R \leftarrow R - 1$
Iteração 2	$i \leftarrow 2,\ s \leftarrow \min\{3,5,6,10\} = 3,\ p[2] \leftarrow N_R(3) = 7,\ R \leftarrow R - 3$
Iteração 3	$i \leftarrow 3,\ s \leftarrow \min\{5,6,10\} = 5,\ p[3] \leftarrow N_R(5) = 4,\ R \leftarrow R - 5$
Iteração 4	$i \leftarrow 4,\ s \leftarrow \min\{6,10\} = 6,\ p[4] \leftarrow N_R(6) = 2,\ R \leftarrow R - 4$
Iteração 5	$i \leftarrow 5,\ s \leftarrow \min\{2,10\} = 2,\ p[5] \leftarrow N_R(2) = 7,\ R \leftarrow R - 2$
Iteração 6	$i \leftarrow 6,\ s \leftarrow \min\{7,10\} = 7,\ p[6] \leftarrow N_R(7) = 8,\ R \leftarrow R - 7$
Iteração 7	$i \leftarrow 7,\ s \leftarrow \min\{8,10\} = 8,\ p[7] \leftarrow N_R(8) = 9,\ R \leftarrow R - 8$
Iteração 8	$i \leftarrow 8,\ s \leftarrow \min\{9,10\} = 9,\ p[8] \leftarrow N_R(9) = 4,\ R \leftarrow R - 9$
Fim	$p = \{6,7,4,2,7,8,9,4\}$

Tabela 4.3: Representação dos passos descritos no Algoritmo 4.1 para o grafo da Figura 4.1.

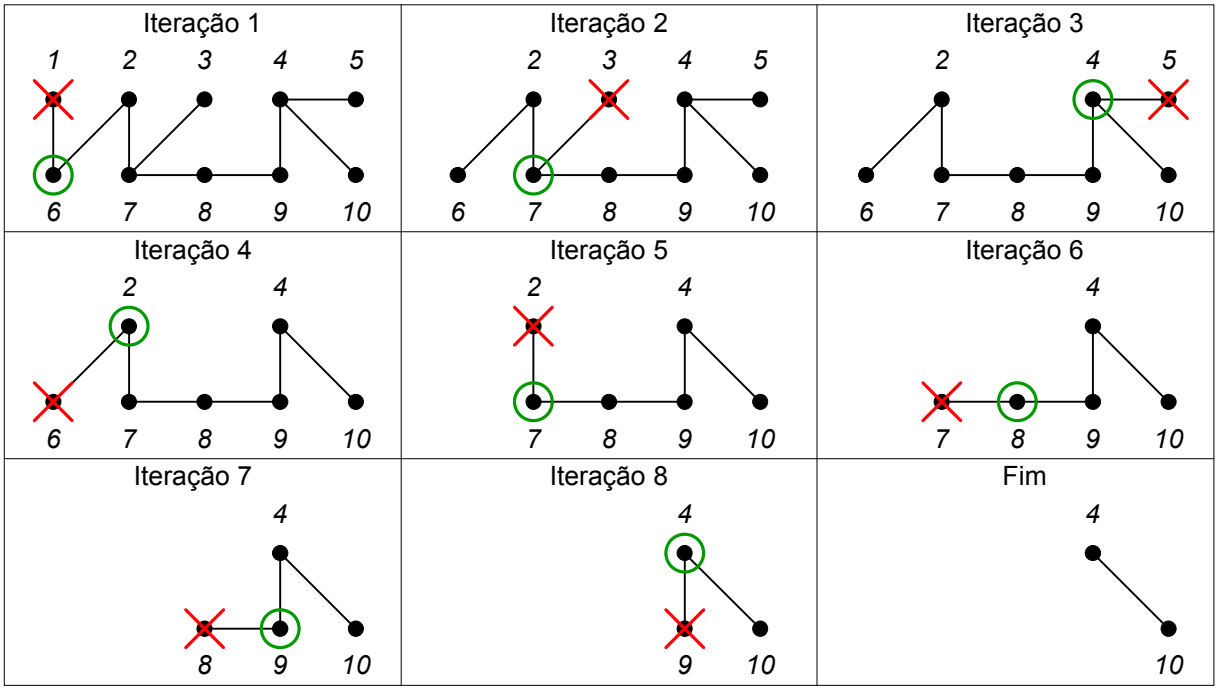

Figura 4.2: Representação gráfica dos passos executados na Tabela 4.3.

Durante a execução de um algoritmo genético é necessário avaliar a aptidão dos cromossomas. Geralmente, as funções de avaliação de aptidão não trabalham directamente com os cromossomas codificados, sendo necessário descodificá-los. O Algoritmo 4.2 [38] permite descodificar os cromossomas, obtendo uma árvore de suporte (Definição 2.10) a

partir de uma sequência de Prüfer, que será exemplificado no Exemplo 4.4.

Algoritmo 4.2: Descodificar uma sequência de Prüfer numa árvore de suporte.

Entrada: $p \rightarrow$ conjunto com a sequência de Prüfer
Saída: $T = (V, E) \rightarrow$ árvore de suporte
Variáveis: $V \rightarrow$ conjunto de vértices da árvore de suporte T
$E \rightarrow$ conjunto de arestas da árvore de suporte T
$n \rightarrow$ número de vértices na árvore de suporte T
$R \rightarrow$ conjunto de vértices temporários
$q \rightarrow$ sequência de Prüfer temporária
$i \rightarrow$ número de iteração actual
$u \rightarrow$ vértice temporário

```
1  n ← |p| + 2
2  V ← {1, 2, ..., n}
3  E ← {}
4  R ← {1, 2, ..., n}
5  q ← p
6  para i ← 1 até n − 2 fazer
7  |  u ← min {R \ q}
8  |  E ← E ∪ {u, p[i]}
9  |  R ← R \ u
10 |  q ← q \ p[i]
11 E ← E ∪ {R[1], R[2]}
12 devolver T = (V, E)
```

Exemplo 4.4.

Descodificar a sequência de Prüfer $p = \{6, 7, 4, 2, 7, 8, 9, 4\}$, representando a árvore de suporte associada a p, recorrendo ao Algoritmo 4.2.

Na Tabela 4.4 são apresentados os passos descritos no Algoritmo 4.2 enquanto que a Figura 4.3 representa graficamente os passos executados na Tabela 4.4. A verde são representadas as arestas adicionadas à árvore de suporte na respectiva iteração.

A sequência de Prüfer p é constituída por oito elementos, dando origem a uma árvore de suporte etiquetada com dez vértices. O conjunto de vértices da árvore de suporte corresponde a $V = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}$. O conjunto de arestas da árvore de suporte corresponde a $E = \{\{1, 6\}, \{3, 7\}, \{5, 4\}, \{6, 2\}, \{2, 7\}, \{7, 8\}, \{8, 9\}, \{9, 4\}, \{4, 10\}\}$. A árvore de suporte obtida, $T = (V, E)$, corresponde à árvore de suporte da Figura 4.1.

□

Início	$n \leftarrow 10,\ V \leftarrow \{1,2,3,4,5,6,7,8,9,10\},\ E \leftarrow \{\},$ $R \leftarrow \{1,2,3,4,5,6,7,8,9,10\},\ q \leftarrow p = \{6,7,4,2,7,8,9,4\}$
Iteração 1	$i \leftarrow 1,\ u \leftarrow \min\{1,3,5,10\} = 1,\ E \leftarrow E \cup \{1,6\},$ $R \leftarrow \{2,3,4,5,6,7,8,9,10\},\ q \leftarrow \{7,4,2,7,8,9,4\}$
Iteração 2	$i \leftarrow 2,\ u \leftarrow \min\{3,5,6,10\} = 3,\ E \leftarrow E \cup \{3,7\},$ $R \leftarrow \{2,4,5,6,7,8,9,10\},\ q \leftarrow \{4,2,7,8,9,4\}$
Iteração 3	$i \leftarrow 3,\ u \leftarrow \min\{5,6,10\} = 5,\ E \leftarrow E \cup \{5,4\},\ R \leftarrow \{2,4,6,7,8,9,10\},$ $q \leftarrow \{2,7,8,9,4\}$
Iteração 4	$i \leftarrow 4,\ u \leftarrow \min\{6,10\} = 6,\ E \leftarrow E \cup \{6,2\},\ R \leftarrow \{2,4,7,8,9,10\},$ $q \leftarrow \{7,8,9,4\}$
Iteração 5	$i \leftarrow 5,\ u \leftarrow \min\{2,10\} = 2,\ E \leftarrow E \cup \{2,7\},\ R \leftarrow \{4,7,8,9,10\},$ $q \leftarrow \{8,9,4\}$
Iteração 6	$i \leftarrow 6,\ u \leftarrow \min\{7,10\} = 7,\ E \leftarrow E \cup \{7,8\},\ R \leftarrow \{4,8,9,10\},\ q \leftarrow \{9,4\}$
Iteração 7	$i \leftarrow 7,\ u \leftarrow \min\{8,10\} = 8,\ E \leftarrow E \cup \{8,9\},\ R \leftarrow \{4,9,10\},\ q \leftarrow \{4\}$
Iteração 8	$i \leftarrow 8,\ u \leftarrow \min\{9,10\} = 9,\ E \leftarrow E \cup \{9,4\},\ R \leftarrow \{4,10\},\ q \leftarrow \{\}$
Fim	$E \leftarrow E \cup \{4,10\},\ T = (V,E)$

Tabela 4.4: Representação dos passos descritos no Algoritmo 4.2 para a sequência de Prüfer p.

Nos próximos parágrafos são apresentadas as propriedades respeitadas e não respeitadas pela codificação por sequências de Prüfer para o problema da árvore de suporte sem restrições.

Os algoritmos de codificação e descodificação de sequências de Prüfer, na sua grande maioria, são baseados no algoritmo apresentado em 1978 por Nijenhuis e Wilf [29]. Os Algoritmos 4.1 e 4.2 são baseados nesse algoritmo, que foi o primeiro algoritmo de tempo linear para as sequências de Prüfer, logo respeitam a propriedade da *eficiência*.

O conjunto $\mathcal{T}(K_n)$, definido na Secção 2.2, denota o conjunto de todas as árvores de suporte para o grafo K_n, tendo cardinalidade definida pela *Fórmula de Cayley*, como referido no teorema 4.1. Assim um grafo completo com n vértices tem n^{n-2} árvores de suporte associadas, que correspondem a soluções na região de admissibilidade para um algoritmo genético. Algumas dessas soluções são não admissíveis para grafos não completos.

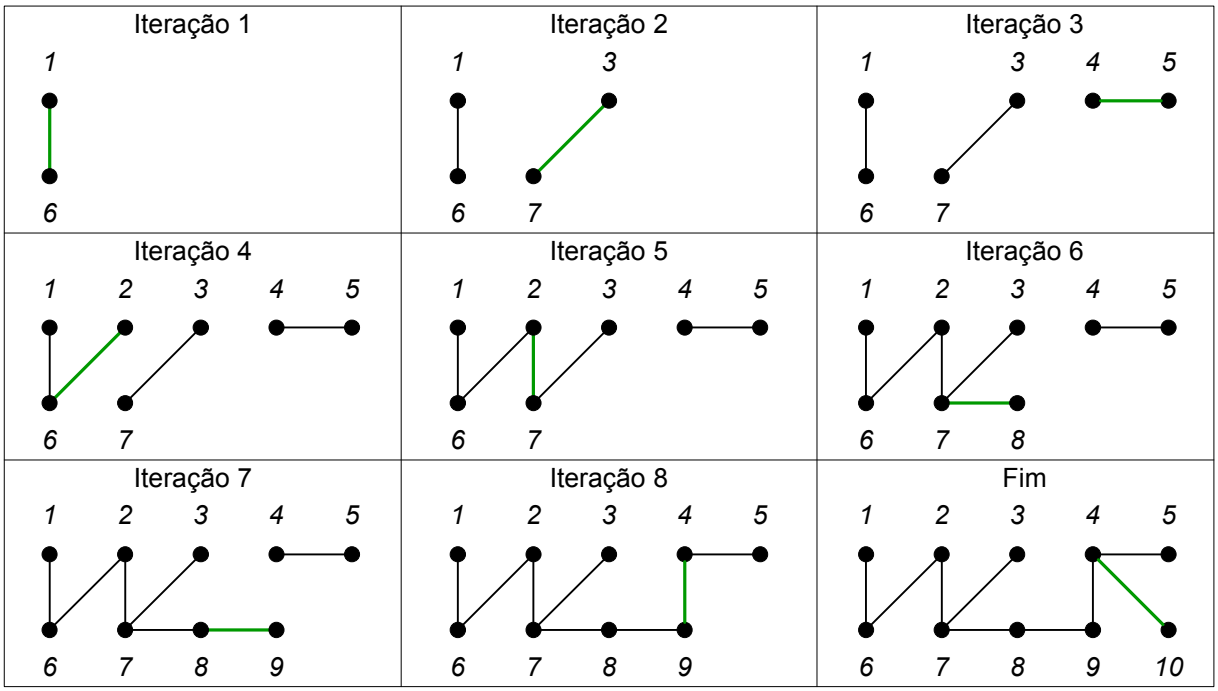

Figura 4.3: Representação gráfica dos passos executados na Tabela 4.4.

Considerando um grafo completo, todas as soluções do espaço de pesquisa têm uma codificação associada e como tal a codificação por sequências de Prüfer respeita a propriedade da *completude*.

A demonstração do Teorema 4.1 (ver [4]) permite estabelecer uma relação de *um-para-um* (bijecção) entre uma árvore de suporte e a correspondente sequência de Prüfer, verificando a propriedade da *não redundância*.

A codificação por sequências de Prüfer não respeita a propriedade da *causalidade* [30], isto é, as variações operadas no cromossoma não são directamente proporcionais às alterações na árvore de suporte correspondente. Note-se que existem sequências de Prüfer que exibem causalidade forte. Estas sequências representam árvores de suporte em que $n-1$ vértices são adjacentes a um outro vértice e são chamadas de *estrelas* [4]. As estrelas são codificadas por $n-2$ etiquetas iguais. No outro extremo estão as sequências constituídas por $n-2$ etiquetas diferentes, chamadas *listas* [4]. As restantes sequências de Prüfer, situam-se entre estes dois extremos. A Figura 4.4 exemplifica uma lista, T_4, uma árvore intermédia, T_5, e uma estrela, T_6.

A Figura 4.5 representa duas sequências de Prüfer. A sequência da esquerda, T_3, representa a sequência original, enquanto que a sequência da direita, T_7, representa a sequência anterior após a mutação de apenas uma etiqueta. Facilmente identificamos apenas três arestas em comum entre ambas as árvores de suporte, sendo estas $\{\{2,7\},\{7,8\},\{8,9\}\}$, embora as sequências difiram apenas numa etiqueta.

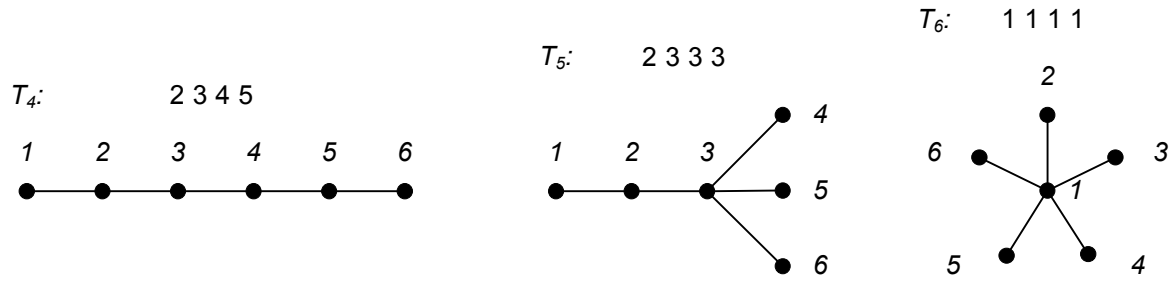

Figura 4.4: Representação de uma lista, de uma árvore intermédia e de uma estrela.

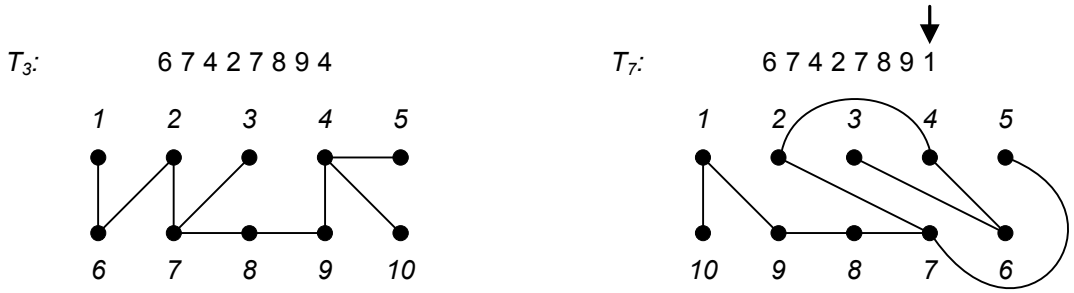

Figura 4.5: Comparação entre duas árvores de suporte, nas quais as suas sequências de Prüfer diferem apenas numa etiqueta.

Sabendo que qualquer sequência de Prüfer, com $n-2$ etiquetas, representa árvores de suporte desde uma estrela até uma lista, inclusive, então qualquer permutação numa etiqueta dessa sequência, respeitando os n vértices da árvore de suporte, continua a representar este tipo de grafos. Desta forma a codificação por sequências de Prüfer respeita a propriedade da *legalidade*.

A propriedade da *hereditariedade*, no contexto das árvores de suporte, indica que os filhos obtidos por cruzamento dos pais, devem representar árvores de suporte com estruturas semelhantes às dos pais. Esta propriedade está associada ao tipo de *causalidade* exibido pela codificação. Assim a *hereditariedade* varia entre fraca para as listas e forte

para as estrelas. Por outro lado, os operadores de cruzamento associados às sequências de Prüfer, tais como o *K-Point Crossover* e o *Uniform Crossover*, não preservam as estruturas dos pais, logo a codificação por sequências de Prüfer não respeita a propriedade da *hereditariedade* [15].

Descritas as propriedades respeitadas, ou não, pela codificação por sequências de Prüfer, constatam-se algumas vantagens e desvantagens no uso desta codificação. Ao nível das vantagens, salientam-se os seguintes pontos:

- esta codificação permite o uso dos operadores genéticos comuns apresentados nas Secções 3.5 e 3.6;
- é uma das codificações mais usadas na literatura e apresenta bons resultados para problemas com poucos vértices [15];
- permite gerar, aleatoriamente, árvores de suporte para as quais a sua descodificação torna-se mais eficiente em relação à geração de uma árvore de suporte pela adição aleatória de arestas, pois neste caso é necessário verificar a formação de ciclos.

Ao nível das desvantagens salientam-se os seguintes pontos:

- apenas uma pequena parte do espaço de pesquisa apresenta causalidade e hereditariedade forte;
- a performance do algoritmo genético tende a diminuir rapidamente com o aumento do número de vértices [15];
- esta codificação não pode ser aplicada directamente a problemas com árvores de suporte com restrições ou cujo grafo base não seja completo, pois quer o cruzamento, quer a mutação, podem gerar soluções não admissíveis (nestes casos são necessários *mecanismos adicionais* para utilizar esta codificação, como referido na Secção 5.2.1).

Alguns autores [15] referem que a codificação de Prüfer é uma má representação das árvores de suporte porque não apresentam as propriedades da causalidade e da hereditariedade, e porque a performance do algoritmo genético é fortemente afectada pelo aumento da ordem do problema.

4.3 Codificação por Sequências de Arestas

A codificação por sequências de arestas (*edge-sets*) foi apresentada em 2000 por Raidl [32] e revista em 2003 por Raidl e Julstrom [34] para o problema da *Árvore de Suporte de Custo Mínimo com Restrições de Grau*. Nesta codificação, um cromossoma é directamente representado pelo conjunto de arestas da árvore de suporte correspondente.

O termo *directamente*, utilizado na afirmação anterior, pode parecer estranho por estarmos a falar de um método de codificação de árvores de suporte. No entanto é perfeitamente justificável como se demonstra de seguida. Na Figura 4.6 está representada a mesma árvore de suporte da Figura 4.1, T_3, conjuntamente com os seus conjuntos de vértices e de arestas. O cromossoma associado à árvore de suporte T_3 corresponde ao conjunto de arestas de T_3, $E(T_3) = \{\{1,6\},\{2,6\},\{2,7\},\{3,7\},\{4,5\},\{4,9\},\{4,10\},\{7,8\},\{8,9\}\}$, justificando o termo *directamente* e, analogamente para a árvore de suporte T_8 também representada na Figura 4.6. Como consequência desta codificação *directa*, não são necessários algoritmos para codificar e descodificar sequências de arestas.

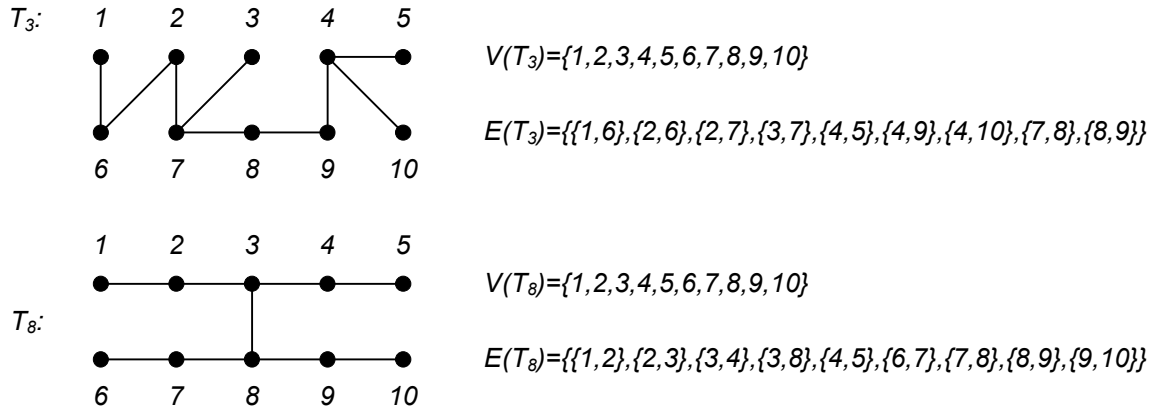

Figura 4.6: Representação de duas árvores de suporte com os seus conjuntos de vértices e de arestas.

A codificação por sequências de Prüfer permite a utilização dos operadores genéticos comuns, apresentados nas Secções 3.5 e 3.6. A Figura 4.7 demonstra o oposto em relação à codificação por sequências de arestas. Na Figura 4.7 são cruzados os cromossomas referentes às árvores de suporte T_3 e T_8 da Figura 4.6, utilizando o operador de cruzamento *One Point Crossover*, com ponto de corte entre o oitavo e o nono locus.

O operador de cruzamento *One Point Crossover* gera dois filhos, F_1 e F_2, também representados na Figura 4.7. Facilmente observamos, em ambos os filhos, a formação de ci-

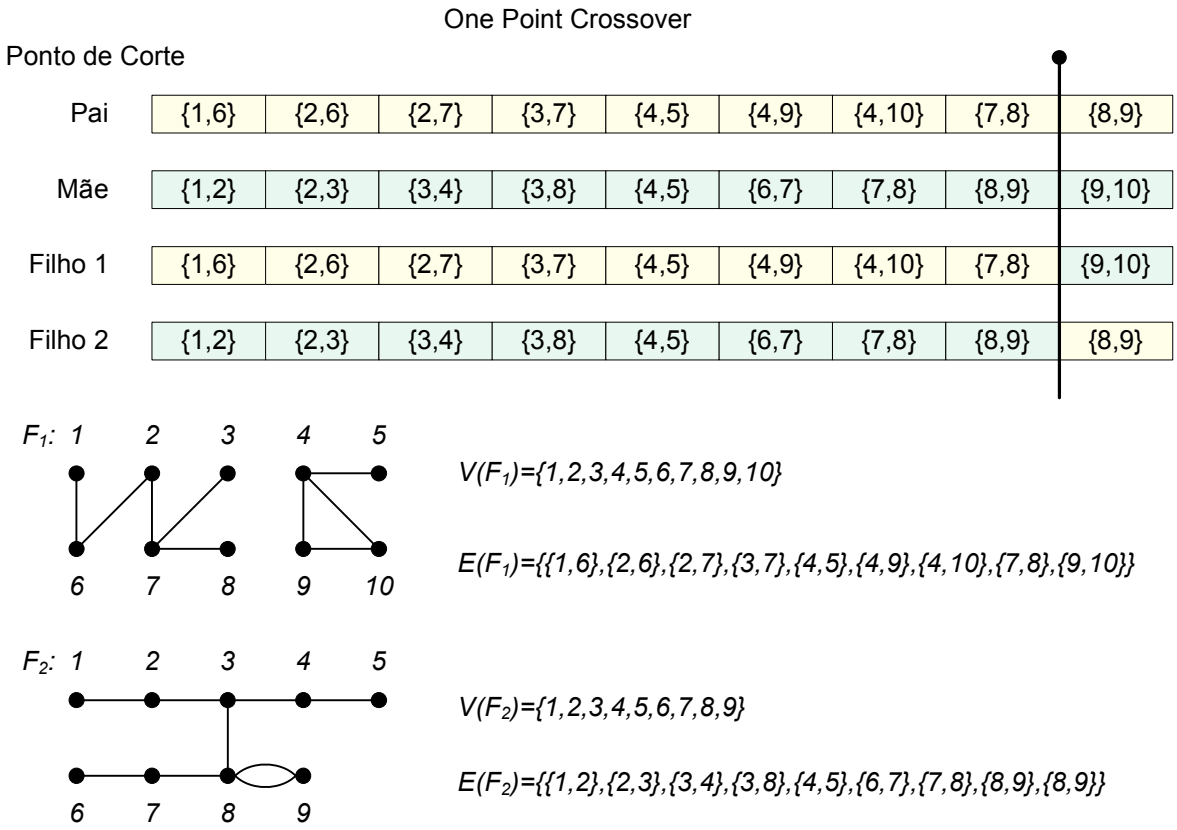

Figura 4.7: Representação da operação de cruzamento utilizando o *One Point Crossover* para dois cromossomas codificados por sequências de arestas.

clos e desconexão das árvores. Os filhos F_1 e F_2 não são árvores de suporte (Definição 2.10) e consequentemente os operadores genéticos comuns não são aplicáveis à codificação por sequências de arestas. Para contornar esta situação, foram apresentados operadores genéticos específicos para esta codificação [34], tendo sido propostos três algoritmos de cruzamento e duas estratégias de mutação. Os algoritmos de cruzamento propostos designam-se por *PrimRST*, *KruskalRST* e *RandWalkRST* (*Random Spanning Tree*≡*RST*). Estes três algoritmos não trabalham directamente com os cromossomas pai e mãe [32]. Primeiro é gerado um grafo auxiliar que contém todos os vértices e arestas dos cromossomas pai e mãe. Posteriormente os algoritmos geram, a partir deste grafo auxiliar, uma árvore de suporte aleatória.

Na Figura 4.8 temos as duas árvores de suporte T_3 e T_8, que correspondem aos cromossomas pai e mãe, respectivamente. Procede-se à união das árvores de suporte gerando o grafo auxiliar $G_3 = T_3 \cup T_8$. A este grafo auxiliar, G_3, é aplicado um dos três algoritmos

anteriores, *RST*, e gerada a árvore de suporte aleatória.

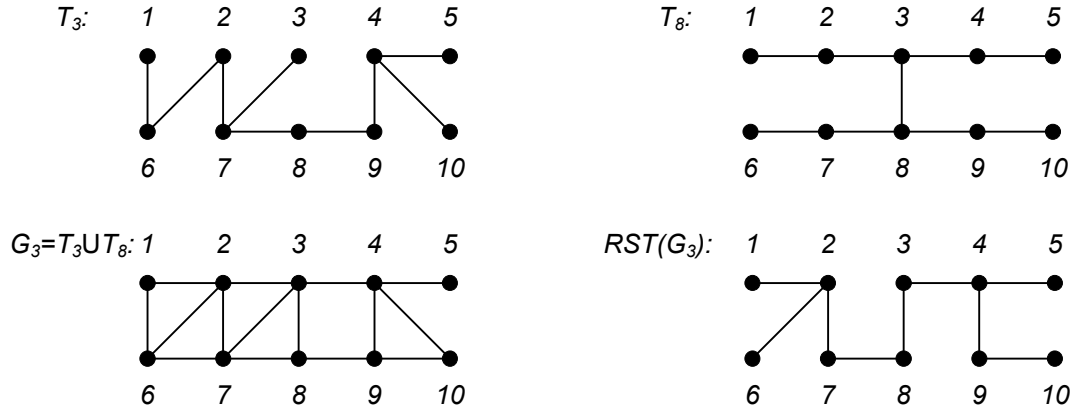

Figura 4.8: Representação da operação de cruzamento utilizando um algoritmo *RST*.

Os algoritmos *PrimRST* e *KruskalRST* baseiam-se, respectivamente, nos algoritmos de *Prim* e de *Kruskal* de determinação da *Árvore de Suporte de Custo Mínimo*. Os algoritmos de *Prim* e de *Kruskal*, podem ser consultados em [4, 38]. Ambos os algoritmos *RST* geram árvores de suporte aleatórias não uniformes, isto é, existe uma maior probabilidade de ser gerada uma árvore de suporte mais semelhante a uma estrela do que a uma lista.

O algoritmo *RandWalkRST* baseia-se num passeio aleatório para gerar árvores de suporte aleatórias. Este algoritmo não possui o problema dos métodos anteriores.

A primeira estratégia de mutação [34] consiste em adicionar, à árvore de suporte, uma aresta do grafo auxiliar. A introdução desta aresta na árvore de suporte vai gerar um ciclo. Neste ciclo é escolhida, aleatoriamente, uma aresta (exceptuando a aresta introduzida) para ser eliminada. Este processo é exemplificado na Figura 4.9, onde a aresta introduzida é representada a verde e as arestas que podem ser eliminadas são representadas a vermelho.

A segunda estratégia de mutação [34] consiste em eliminar aleatoriamente uma aresta da árvore de suporte, gerando um grafo desconexo. Para voltar a ter uma árvore de suporte, é necessário introduzir uma aresta do grafo auxiliar (diferente da aresta eliminada), de forma a tornar o grafo conexo. A Figura 4.10 exemplifica esta estratégia de mutação, na qual a aresta a eliminar é representada a vermelho e a aresta a adicionar é representada a verde. Note-se que as Figuras 4.8, 4.9 e 4.10 são baseadas nas figuras 5 e 6 de [34].

A codificação por sequências de arestas respeita todas as propriedades apresentadas

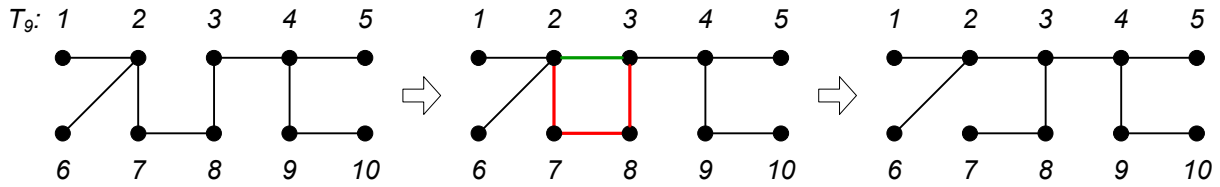

Figura 4.9: Representação da primeira estratégia de mutação para a codificação por sequências de arestas.

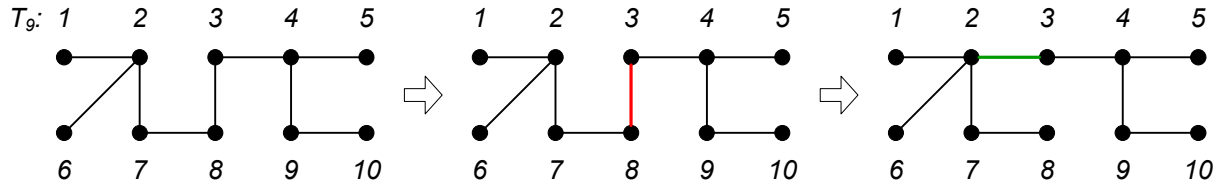

Figura 4.10: Representação da segunda estratégia de mutação para a codificação por sequências de arestas.

na Secção 4.1 para o problema da árvore de suporte sem restrições. A propriedade da *eficiência* é garantida pois não são necessários algoritmos para codificar e descodificar as sequências de arestas. As propriedades da *completude* e da *não redundância* são verificadas, pois o número de soluções que se podem representar em ambas as componentes (dos *genótipos* e dos *fenótipos*) é o mesmo, além de existir uma relação unívoca entre os cromossomas e as árvores de suporte. Considerando os operadores genéticos de cruzamento (*PrimRST*, *KruskalRST* e *RandWalkRST*) e as duas estratégias de mutação, podemos concluir também que a codificação por sequências de arestas respeita a propriedade da *legalidade*. Os operadores genéticos, atrás mencionados, respeitam a propriedade da *causalidade* [32], tendo causalidade forte, e consequentemente, é verificada a propriedade da *hereditariedade*, com hereditariedade forte.

Podemos salientar as seguintes vantagens para a codificação por sequências de arestas:

- não necessita de algoritmos para codificar e descodificar uma sequência de arestas, permitindo aumentar a performance do algoritmo genético;
- respeita todas as propriedades propostas para as codificações, tendo causalidade e hereditariedade fortes;
- permite adicionar, facilmente, aos operadores genéticos especializados, restrições a serem respeitadas pelo problema [34].

Ao nível das desvantagens, salientam-se os seguintes pontos:

- não suporta os operadores genéticos comuns;
- necessita de operadores genéticos especializados que consomem mais tempo de processamento em relação aos operadores genéticos comuns;
- alguns dos operadores genéticos especializados tendem a gerar árvores de suporte sem distribuição uniforme (mais próximas das estrelas do que das listas), podendo influenciar a convergência do algoritmo genético.

O algoritmo do método de cruzamento *PrimRST* será descrito e exemplificado no Capítulo 5, assim como as propriedades respeitadas e não respeitadas para o problema da *Árvore de Suporte de Custo Mínimo com Restrições de Salto*, para ambas as codificações em estudo.

Capítulo 5

Implementação

Neste capítulo é apresentado o esboço da implementação do algoritmo genético para o problema da *Árvore de Suporte de Custo Mínimo com Restrições de Salto*.

O algoritmo implementado compreende três módulos:

1. módulo de grafos;
2. módulo do algoritmo genético;
3. módulo de testes.

Estes módulos foram implementados na linguagem de programação Java recorrendo ao livro [3]. Existem bibliotecas nesta linguagem que permitem uma fácil utilização de grafos, por exemplo, *Java Data Structures Library*, e de algoritmos genéticos, por exemplo, *Java Genetic Algorithms Package* ou *Java API for Genetic Algorithms*. No entanto estas bibliotecas não possuem a flexibilidade que se pretendia para um funcionamento em conjunto, por exemplo conjugar a utilização da biblioteca *Java Data Structures Library* com a biblioteca *Java Genetic Algorithms Package*. No âmbito de aprofundar o estudo sobre os algoritmos genéticos, foi decidido implementar, os módulos mencionados, de raiz.

5.1 Módulo de Grafos

No *módulo de grafos* são implementados os *objectos** grafo, vértice e aresta, assim como os procedimentos que manipulam e efectuam operações sobre estes objectos. São definidas as operações que permitem adicionar, remover e pesquisar vértices e arestas num grafo, assim como as operações que permitem verificar adjacências, incidências, dimensão

e ordem de um grafo, entre outras apresentadas na Secção 2.1. Para garantir a eficiência das operações mencionadas, é utilizada uma tabela de dispersão (*Hash Table*) [3] para armazenar as adjacências. As tabelas de dispersão permitem adicionar, remover e pesquisar elementos em tempo que varia entre constante e linear, sendo uma das estruturas de dados mais eficientes da linguagem Java [3]. Esta eficiência da tabela de dispersão constitui uma vantagem para o algoritmo genético, porque grande parte do tempo de processamento de um algoritmo genético consiste em verificar se os filhos gerados por cruzamento e/ou mutação são admissíveis, implicando um grande número de pesquisas para comparar as arestas da árvore gerada com as arestas do problema.

Neste módulo são também implementados os mecanismos que permitem a leitura de um grafo completo a partir de um ficheiro e no qual se pretende determinar uma *Árvore de Suporte de Custo Mínimo com Restrições de Salto*. Note-se que para todos os problemas, o vértice raiz consiste no último vértice da matriz de adjacência.

Outras operações implementadas neste módulo consistem em operações que permitem obter os vértices vizinhos de um vértice (Definição 2.4), obter o grau de um vértice (Definição 2.3), definir uma árvore de suporte de um grafo (Definição 2.10), definir um vértice raiz e calcular o salto associado a cada vértice a partir desse vértice raiz. O algoritmo que calcula o salto associado a cada vértice considerando um vértice raiz consiste em atribuir ao vértice raiz o salto com o valor 0. O algoritmo inicia-se no vértice raiz, determinando a lista de vértices adjacentes (vizinhos) ao vértice raiz. O salto desses vértices define-se como o salto do vértice raiz incrementado em uma unidade. Posteriormente, determina-se a lista de vértices adjacentes aos vértices actuais e o seu salto corresponde ao salto actual incrementado em uma unidade, e assim sucessivamente.

5.2 Módulo do Algoritmo Genético

No *módulo do algoritmo genético* são implementados o motor do algoritmo genético, juntamente com os métodos de geração, selecção, cruzamento, mutação e renovação de indivíduos da população, apresentados no Capítulo 3, e as codificações de cromossomas apresentadas no Capítulo 4, específicas para a resolução do problema da *Árvore de Suporte de Custo Mínimo com Restrições de Salto*, apresentado na Secção 2.3.

*"Um objecto é um elemento, auto-contido e com um propósito específico, de um programa de computador, que representa um conjunto de propriedades relacionadas. Os objectos interagem entre si de forma altamente controlada" [3].

O algoritmo genético implementado segue os 11 passos apresentados no diagrama da Figura 5.1 e descritos de seguida.

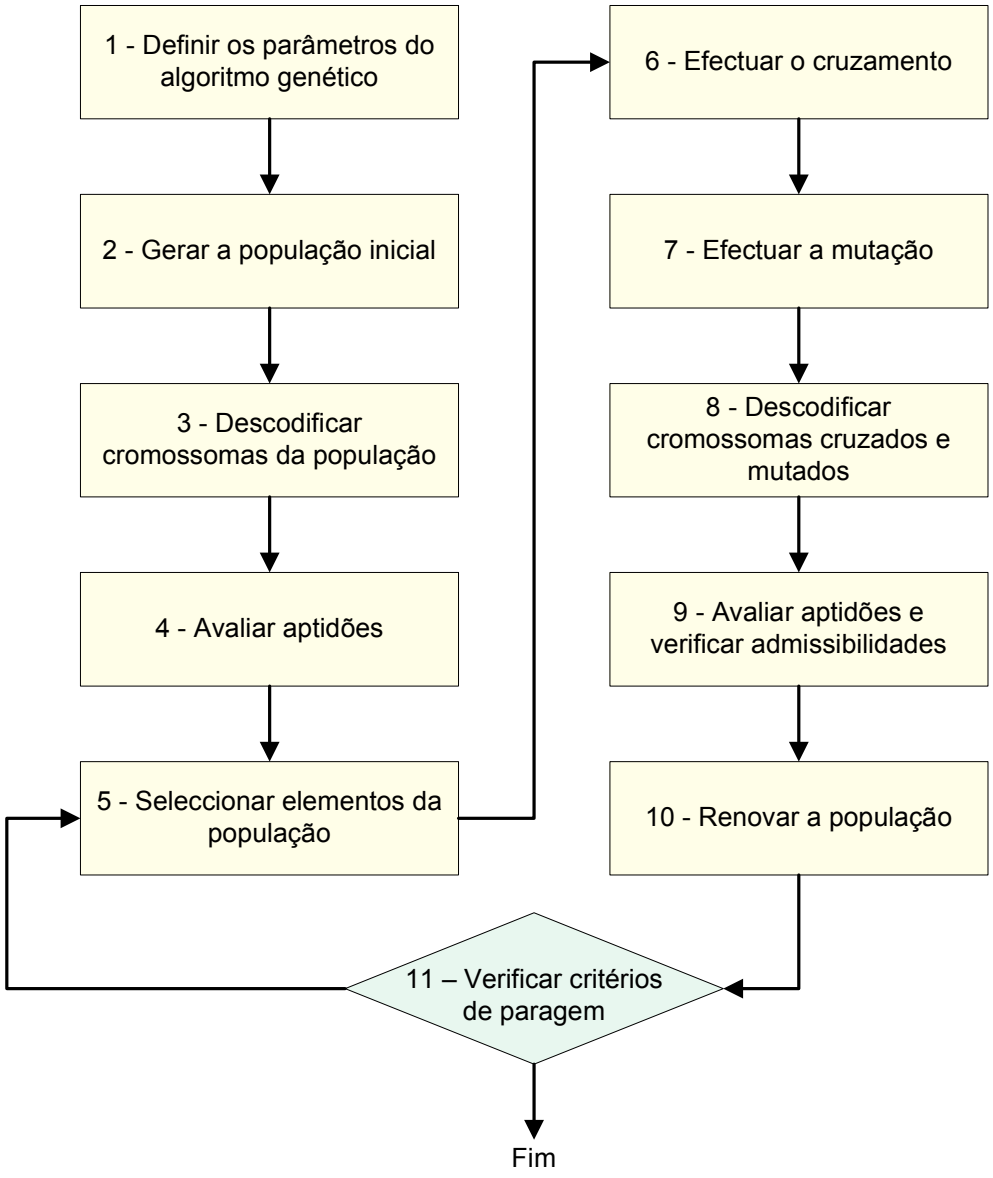

Figura 5.1: Diagrama do algoritmo genético implementado.

Passo 1

O primeiro passo consiste em definir os parâmetros do algoritmo genético. Estes parâmetros influenciam a eficiência do algoritmo genético, podendo fazê-lo convergir rapidamente para uma solução possivelmente óptima local, ou permitir uma melhor exploração do espaço de pesquisa com a diversificação dos indivíduos da população.

Os parâmetros do algoritmo genético são:

- o número máximo de iterações do algoritmo genético;
- a dimensão da população;
- a dimensão de um torneio;
- o número de torneios;
- a percentagem de mutação;
- o número de iterações para renovação da população.

O impacto destes parâmetros na eficiência do algoritmo genético será apresentado na Secção 6.2. Cada parâmetro será descrito nos respectivos passos do algoritmo genético.

Passo 2

O segundo passo consiste em gerar a população inicial do algoritmo genético, isto é, consiste em gerar um conjunto de cromossomas admissíveis para o problema da *Árvore de Suporte de Custo Mínimo com Restrições de Salto*, numa das duas codificações em estudo, a codificação por sequências de Prüfer ou a codificação por sequências de arestas.

Para gerar um indivíduo foram considerados dois métodos, um completamente aleatório e outro baseado numa heurística que respeita a restrição de salto.

Considerando que o problema tem ordem n, o primeiro método consiste em gerar sequências aleatórias de números de inteiros com $n-2$ elementos (Teorema 4.1). Estes cromossomas gerados aleatoriamente correspondem a sequências com codificação de Prüfer, que facilmente se convertem em sequências de arestas, caso a codificação em uso seja essa. O primeiro método não garante que os indivíduos gerados respeitem a restrição de salto, logo muitos desses indivíduos gerados têm de ser eliminados, representando um impacto negativo a nível do tempo de processamento.

Para contornar esta situação, foi implementada uma heurística que permite gerar indivíduos sempre admissíveis, isto é, árvores de suporte que respeitam a restrição de salto. A heurística consiste no seguinte. O nível 0 da árvore de suporte é composto apenas pelo vértice raiz. Nos restantes níveis são calculados aleatoriamente o número de vértices a adicionar a esse nível, sabendo que cada nível tem pelo menos um vértice. No último nível são adicionados todos os vértices que não foram adicionados nos níveis anteriores (existe pelo menos um vértice no último nível). Por outro lado, são apenas permitidas arestas entre vértices de níveis imediatamente anterior e posterior, com a excepção do nível 0 que é apenas adjacente ao nível 1, e do último nível que é apenas adjacente ao nível precedente, evitando a formação de ciclos. Os passos executados pela heurística *HRST* (*Hop-constrained*

Random Spanning Tree heuristic) estão descritos no Algoritmo 5.1 e exemplificados no Exemplo 5.1.

Algoritmo 5.1: Heurística *HRST* de geração de um indivíduo árvore, respeitando o valor de salto H da *Árvore de Suporte de Custo Mínimo com Restrições de Salto*.

Entrada: $G = (V, E) \rightarrow$ grafo completo
$r \rightarrow$ vértice raiz
$H \rightarrow$ número máximo de saltos permitido
Saída: $T = (C, F) \rightarrow$ árvore de suporte
Variáveis: $F \rightarrow$ conjunto de arestas da árvore de suporte T
$C \rightarrow$ conjunto de vértices da árvore de suporte T
$R \rightarrow$ conjunto de vértices disponíveis para incluir na árvore de suporte T
$N \rightarrow$ conjunto de vértices no nível anterior ao nível actual
$S \rightarrow$ conjunto de vértices no nível actual
$i \rightarrow$ nível actual
$m \rightarrow$ número máximo de vértices possíveis de adicionar no nível i
$o \rightarrow$ número de vértices a adicionar no nível i
$j \rightarrow$ vértice a adicionar no nível i
$u, v \rightarrow$ vértices temporários

```
1  R ← V \ r
2  N ← {r}
3  F ← {}
4  C ← {r}
5  para i ← 1 até H fazer
6      m ← |R| − (H − i)
7      se i < H então //calcular o número de vértices a adicionar num nível
       inferior a H
8          escolher aleatoriamente um valor o entre 1 e m
9      senão //calcular o número de vértices a adicionar no nível H
10         o ← |R| //restantes vértices que faltam adicionar
11     S ← {}
12     para j ← 1 até o fazer //adicionar arestas entre os vértices do nível
       anterior e os vértices do nível actual
13         escolher aleatoriamente um vértice u ∈ N
14         escolher aleatoriamente um vértice v ∈ R
15         F ← F ∪ {u, v}
16         R ← R \ v
17         S ← S ∪ v
18     N ← S
19     C ← C ∪ N
20 devolver T = (C, F)
```

Este algoritmo gera um indivíduo devolvendo a árvore de suporte correspondente, que posteriormente é codificada numa das duas codificações em estudo. A codificação por sequências de Prüfer recorre ao Algoritmo 4.1, apresentado na Secção 4.2, enquanto que a codificação por sequências de arestas consiste em considerar o conjunto de arestas da árvore de suporte gerada, tal como apresentado na Secção 4.3.

Para gerar uma população de indivíduos com uma determinada codificação, é executado repetidamente o método aleatório ou o método heurístico (Algoritmo 5.1), seguido do algoritmo da respectiva codificação, até que o número de indivíduos gerados iguale o valor do parâmetro *dimensão da população*.

O Algoritmo 5.1 usa os conceitos de grafo completo (Secção 2.1), árvore de suporte (Definição 2.10) e de salto (Definição 2.13) apresentados no Capítulo 2.

Exemplo 5.1.

Geração de um indivíduo em ambas as codificações em estudo, recorrendo ao Algoritmo 5.1, considerando o problema representado na Figura 5.2.

Consideremos o problema representado pelo grafo K_6 da Figura 5.2, o valor de salto $H = 3$ e o vértice raiz representado pelo vértice 0. A Tabela 5.1 apresenta os passos executados pelo Algoritmo 5.1, para o ciclo exterior com índices i e para o ciclo interior com índices j. Na Figura 5.3 são representados graficamente os passos executados na Tabela 5.1. A verde são representadas as arestas adicionadas à árvore de suporte na respectiva iteração.

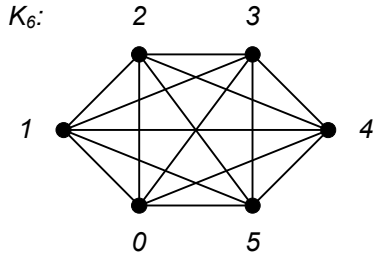

Figura 5.2: Representação de um grafo completo de ordem 6, K_6.

O algoritmo da heurística *HRST* é constituído por dois ciclos, um exterior e outro interior. O ciclo exterior, etiquetado pelos índices i, indica o nível actual da árvore de suporte, que neste exemplo está limitada a três níveis, $H = 3$. O ciclo interior, etiquetado pelos índices j, é responsável por adicionar arestas entre o nível actual e o nível anterior.

Na primeira iteração calculamos o número máximo de vértices que podemos adicionar no nível 1, $m = 3$. Como o nível actual é inferior ao nível máximo permitido, H, então o número de vértices a adicionar neste nível, o, corresponde a um número aleatório entre 1 e m. O passo seguinte consiste em seleccionar vértices e adicionar arestas entre esses vértices e os vértices do nível anterior. Consideremos $o = 1$, logo será adicionado um vértice no nível 1. Seja $v = 3$ o vértice seleccionado para adicionar no nível 1, de entre os vértices possíveis, R. Adicionamos uma aresta entre o vértice raiz e o vértice $v = 3$, retiramos o vértice v de R e adicionamos o mesmo vértice ao conjunto de vértices do nível actual, S.

Na segunda iteração os vértices do conjunto S passam a representar os vértices do nível anterior, N. Neste nível serão adicionados dois vértices, valor aleatório obtido pela actualização dos valores de m e o. Os vértices seleccionados aleatoriamente de R, 1 e 4, são ligados ao único vértice do nível anterior, o vértice 3.

Na terceira iteração os vértices do nível anterior, N, são os vértices 1 e 4. Neste nível somos obrigados a adicionar os restantes vértices do conjunto R, 2 e 5. Estes vértices apenas podem ser ligados aos vértices do nível anterior, 1 e 4. Em ambos os casos o vértice do nível anterior escolhido aleatoriamente foi o vértice 4.

O indivíduo gerado tem a seguinte codificação por sequências de Prüfer $\{3, 3, 4, 4\}$ e tem a seguinte codificação por sequências de arestas $\{\{0, 3\}, \{3, 1\}, \{3, 4\}, \{4, 2\}, \{4, 5\}\}$.

Início	$R \leftarrow \{1, 2, 3, 4, 5\},\ N \leftarrow \{0\},\ F \leftarrow \{\},\ C \leftarrow \{0\}$
Iteração 1	$i \leftarrow 1,\ m \leftarrow 5 - (3 - 1) = 3,\ i < H \Leftrightarrow 1 < 3,\ o \leftarrow 1,\ S \leftarrow \{\},$ $j \leftarrow 1,\ u \leftarrow 0,\ v \leftarrow 3,\ F \leftarrow F \cup \{0, 3\},\ R \leftarrow \{1, 2, 4, 5\},\ S \leftarrow \{3\},$ $N \leftarrow \{3\},\ C \leftarrow \{0, 3\}$
Iteração 2	$i \leftarrow 2,\ m \leftarrow 4 - (3 - 2) = 3,\ i < H \Leftrightarrow 2 < 3,\ o \leftarrow 2,\ S \leftarrow \{\},$ $j \leftarrow 1,\ u \leftarrow 3,\ v \leftarrow 1,\ F \leftarrow F \cup \{3, 1\},\ R \leftarrow \{2, 4, 5\},\ S \leftarrow \{1\},$ $j \leftarrow 2,\ u \leftarrow 3,\ v \leftarrow 4,\ F \leftarrow F \cup \{3, 4\},\ R \leftarrow \{2, 5\},\ S \leftarrow \{1, 4\},$ $N \leftarrow \{1, 4\},\ C \leftarrow \{0, 3, 1, 4\}$
Iteração 3	$i \leftarrow 3,\ m \leftarrow 2 - (3 - 3) = 2,\ i \not< H \Leftrightarrow 3 \not< 3,\ o \leftarrow \lvert R \rvert = 2,\ S \leftarrow \{\},$ $j \leftarrow 1,\ u \leftarrow 4,\ v \leftarrow 2,\ F \leftarrow F \cup \{4, 2\},\ R \leftarrow \{5\},\ S \leftarrow \{2\},$ $j \leftarrow 2,\ u \leftarrow 4,\ v \leftarrow 5,\ F \leftarrow F \cup \{4, 5\},\ R \leftarrow \{\},\ S \leftarrow \{2, 5\},$ $N \leftarrow \{2, 5\},\ C \leftarrow \{0, 3, 1, 4, 2, 5\}$
Fim	$T = (C, F)$

Tabela 5.1: Representação dos passos descritos no Algoritmo 5.1 para o grafo da Figura 5.2.

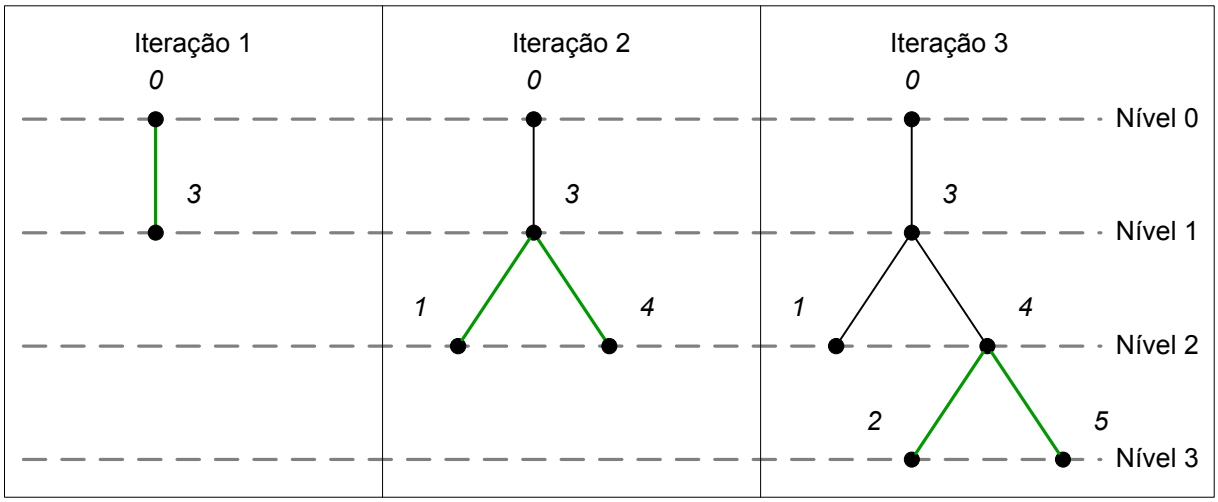

Figura 5.3: Representação gráfica dos passos executados na Tabela 5.1.

□

O cromossoma associado à codificação por sequências de Prüfer corresponde a um vector de inteiros, onde cada posição do vector contém um valor inteiro, que será exemplificado na Figura 5.4.

O cromossoma associado à codificação por sequências de arestas corresponde a um vector de *pares de vértices*. Um par de vértices é um *objecto**, definido na linguagem de programação Java, com capacidade para o armazenamento de duas variáveis (vértices), que corresponde a uma aresta. Cada posição do vector do cromossoma contém um par de vértices (aresta), que será também exemplificado na Figura 5.4.

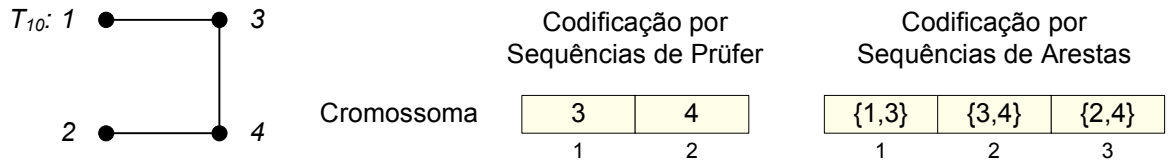

Figura 5.4: Representação de uma árvore de suporte e dos respectivos cromossomas codificados por sequências de Prüfer e por sequências de arestas.

Passo 3

O terceiro passo consiste em descodificar todos os cromossomas da população. Para descodificar um cromossoma com codificação por sequências de Prüfer recorre-se ao Algoritmo 4.2, apresentado na Secção 4.2, enquanto que para descodificar um cromossoma com

codificação por sequências de arestas, é necessário juntar ao cromossoma o conjunto dos vértices presentes nesse cromossoma para obter a árvore de suporte correspondente.

Passo 4

O quarto passo consiste na avaliação da aptidão dos cromossomas da população, descodificados no passo anterior. A avaliação da aptidão consiste em calcular o custo total das arestas que constituem a respectiva árvore de suporte. Além disso, é verificada admissibilidade do cromossoma para o grafo do problema, e se é cumprida a restrição de salto. Os primeiros dois casos estão automaticamente verificados pela heurística apresentada no Algoritmo 5.1, na qual cada cromossoma gerado é admissível para o problema e respeita a restrição de salto do problema. Portanto o quarto passo resume-se a calcular as aptidões de todos os cromossomas da população. Note-se que por uma questão de eficiência, a população passa a estar organizada por aptidão, facilitando o processo de renovação da população (nono passo). O algoritmo de ordenação usado foi o *Gnome Sort*, proposto em 2000 por Hamid Sarbazi-Azad [35].

Passo 5

O quinto passo consiste em seleccionar indivíduos da população para as operações de cruzamento e mutação. A selecção dos indivíduos processa-se de acordo com o método do Torneio, apresentado na Secção 3.4.2, na qual um torneio consiste na escolha aleatória de indivíduos da população, onde é seleccionado o indivíduo mais apto. O carácter aleatório com distribuição uniforme da selecção de indivíduos da população não é penalizado pela organização da população por aptidão, porque cada indivíduo tem a mesma probabilidade de selecção. O método do Torneio é responsável pelo uso de dois dos parâmetros do algoritmo genético implementado, nomeadamente a *dimensão de um torneio* e o *número de torneios*. A *dimensão de um torneio* (dimensão da selecção) corresponde ao número de indivíduos que serão escolhidos aleatoriamente para a realização do torneio, no qual o indivíduo mais apto é seleccionado. O *número de torneios* corresponde ao número de torneios a realizar, sendo no mínimo dois torneios, que correspondem ao número mínimo de progenitores necessários para o passo do cruzamento.

Passo 6

O sexto passo corresponde ao passo do cruzamento. Neste passo os indivíduos seleccionados pelo passo anterior são agrupados em pares de progenitores aos quais é aplicado um método de cruzamento.

Se os indivíduos estão codificados por sequências de Prüfer, o método de cruzamento utilizado é o *One Point Crossover*, apresentado na Secção 3.5.1.

Se os indivíduos estão codificados por sequências de arestas, o método de cruzamento utilizado é o *PrimRST*, referido na Secção 4.3. O método de cruzamento *PrimRST* permite gerar uma árvore de suporte aleatória, a partir de um grafo auxiliar que contém as arestas dos cromossomas pai e mãe. Em cada iteração é seleccionada aleatoriamente uma aresta, de um conjunto de arestas disponíveis. Se essa aresta não formar um ciclo após a introdução na árvore, então é escolhida e adicionada à árvore. O processo repete-se até todos os vértices pertencerem à árvore de suporte. Os passos executados pelo método de cruzamento *PrimRST* estão representados no Algoritmo 5.2 e exemplificados no Exemplo 5.2.

O Algoritmo 5.2 usa os conceitos de grafo conexo (Definição 2.8) e de árvore de suporte (Definição 2.10) apresentados no Capítulo 2.

Algoritmo 5.2: Método de cruzamento *PrimRST*.

Entrada: $G = (V, E) \rightarrow$ grafo auxiliar (conexo)
Saída: $T = (C, F) \rightarrow$ árvore de suporte
Variáveis: $F \rightarrow$ conjunto das arestas da árvore de suporte T
$C \rightarrow$ conjunto dos vértices da árvore de suporte T
$A \rightarrow$ conjunto das arestas temporárias
$s \rightarrow$ vértice inicial
$u, v, w \rightarrow$ vértices temporários

```
1  F ← {}
2  escolher aleatoriamente um vértice inicial s ∈ V
3  C ← {s}
4  A ← {{s, v} ∈ E : v ∈ V}
5  enquanto C ≠ V fazer
6      escolher aleatoriamente uma aresta {u, v} ∈ A, com u ∈ C
7      A ← A \ {u, v}
8      se v ∉ C então //colocar a aresta {u, v} na árvore de suporte
9          F ← F ∪ {u, v}
10         C ← C ∪ v
11         A ← A ∪ {{v, w} ∈ E : w ∉ C}
12 devolver T = (C, F)
```

Exemplo 5.2.

Aplicação do método de cruzamento PrimRST *ao grafo G_3, representado na Figura 5.5, seguindo os passos descritos no Algoritmo 5.2.*

O grafo G_3 foi apresentado na Figura 4.8 da Secção 4.3, representando o grafo auxiliar resultante da união dos cromossomas pai e mãe, T_3 e T_8 respectivamente, também apresentados na Figura 4.8. Neste exemplo é aplicado o método de cruzamento *PrimRST* ao grafo G_3 de modo a obter o grafo $RST\,(G_3)$ (Figura 4.8), que representa uma das várias árvores de suporte aleatórias que se podem obter por este método.

Início	$F \leftarrow \{\}\,,\ s \leftarrow 3,\ C \leftarrow \{3\}\,,\ A \leftarrow \{\{3,2\}\,,\{3,4\}\,,\{3,7\}\,,\{3,8\}\}$
Iteração 1	$\{u,v\} \leftarrow \{3,4\}\,,\ 4 \notin C,\ C \leftarrow \{3,4\}\,,\ F \leftarrow F \cup \{3,4\}\,,$ $A \leftarrow \{\{3,2\}\,,\{3,7\}\,,\{3,8\}\,,\{4,5\}\,,\{4,9\}\,,\{4,10\}\}$
Iteração 2	$\{u,v\} \leftarrow \{4,9\}\,,\ 9 \notin C,\ C \leftarrow \{3,4,9\}\,,\ F \leftarrow F \cup \{4,9\}\,,$ $A \leftarrow \{\{3,2\}\,,\{3,7\}\,,\{3,8\}\,,\{4,5\}\,,\{4,10\}\,,\{9,8\}\,,\{9,10\}\}$
Iteração 3	$\{u,v\} \leftarrow \{3,8\}\,,\ 8 \notin C,\ C \leftarrow \{3,4,9,8\}\,,\ F \leftarrow F \cup \{3,8\}\,,$ $A \leftarrow \{\{3,2\}\,,\{3,7\}\,,\{4,5\}\,,\{4,10\}\,,\{9,8\}\,,\{9,10\}\,,\{8,7\}\}$
Iteração 4	$\{u,v\} \leftarrow \{9,10\}\,,\ 10 \notin C,\ C \leftarrow \{3,4,9,8,10\}\,,\ F \leftarrow F \cup \{9,10\}\,,$ $A \leftarrow \{\{3,2\}\,,\{3,7\}\,,\{4,5\}\,,\{4,10\}\,,\{9,8\}\,,\{8,7\}\}$
Iteração 5	$\{u,v\} \leftarrow \{9,8\}\,,\ 8 \in C,\ A \leftarrow \{\{3,2\}\,,\{3,7\}\,,\{4,5\}\,,\{4,10\}\,,\{8,7\}\}$
Iteração 6	$\{u,v\} \leftarrow \{8,7\}\,,\ 7 \notin C,\ C \leftarrow \{3,4,9,8,10,7\}\,,\ F \leftarrow F \cup \{8,7\}\,,$ $A \leftarrow \{\{3,2\}\,,\{3,7\}\,,\{4,5\}\,,\{4,10\}\,,\{7,2\}\,,\{7,6\}\}$
Iteração 7	$\{u,v\} \leftarrow \{7,2\}\,,\ 2 \notin C,\ C \leftarrow \{3,4,9,8,10,7,2\}\,,\ F \leftarrow F \cup \{7,2\}\,,$ $A \leftarrow \{\{3,2\}\,,\{3,7\}\,,\{4,5\}\,,\{4,10\}\,,\{7,6\}\,,\{2,1\}\,,\{2,6\}\}$
Iteração 8	$\{u,v\} \leftarrow \{4,5\}\,,\ 5 \notin C,\ C \leftarrow \{3,4,9,8,10,7,2,5\}\,,\ F \leftarrow F \cup \{4,5\}\,,$ $A \leftarrow \{\{3,2\}\,,\{3,7\}\,,\{4,10\}\,,\{7,6\}\,,\{2,1\}\,,\{2,6\}\}$
Iteração 9	$\{u,v\} \leftarrow \{3,2\}\,,\ 2 \in C,\ A \leftarrow \{\{3,7\}\,,\{4,10\}\,,\{7,6\}\,,\{2,1\}\,,\{2,6\}\}$
Iteração 10	$\{u,v\} \leftarrow \{2,6\}\,,\ 6 \notin C,\ C \leftarrow \{3,4,9,8,10,7,2,5,6\}\,,\ F \leftarrow F \cup \{2,6\}\,,$ $A \leftarrow \{\{3,7\}\,,\{4,10\}\,,\{6,1\}\,,\{7,6\}\,,\{2,1\}\}$
Iteração 11	$\{u,v\} \leftarrow \{2,1\}\,,\ 1 \notin C,\ C \leftarrow \{3,4,9,8,10,7,2,5,6,1\}\,,\ F \leftarrow F \cup \{2,1\}$ $A \leftarrow \{\{3,7\}\,,\{4,10\}\,,\{6,1\}\,,\{7,6\}\}$
Fim	$T = (C,F)$

Tabela 5.2: Representação dos passos descritos no Algoritmo 5.2 para o método de cruzamento *PrimRST*, considerando o grafo G_3 da Figura 5.5.

A Tabela 5.2 apresenta os passos executados pelo Algoritmo 5.2, *PrimRST*, onde cada iteração representa o estado após a execução dos passos 5 a 11. A tabela inclui duas iterações, a 5 e a 9, onde a aresta seleccionada não pode ser adicionada à árvore de suporte porque formaria um ciclo. Na Figura 5.5 são representados graficamente os passos executados na Tabela 5.2. A verde está representada a aresta adicionada à árvore de suporte na respectiva iteração. A vermelho está representada a aresta que se adicionada à árvore de suporte da respectiva iteração formaria um ciclo, e, consequentemente, não é adicionada à árvore de suporte.

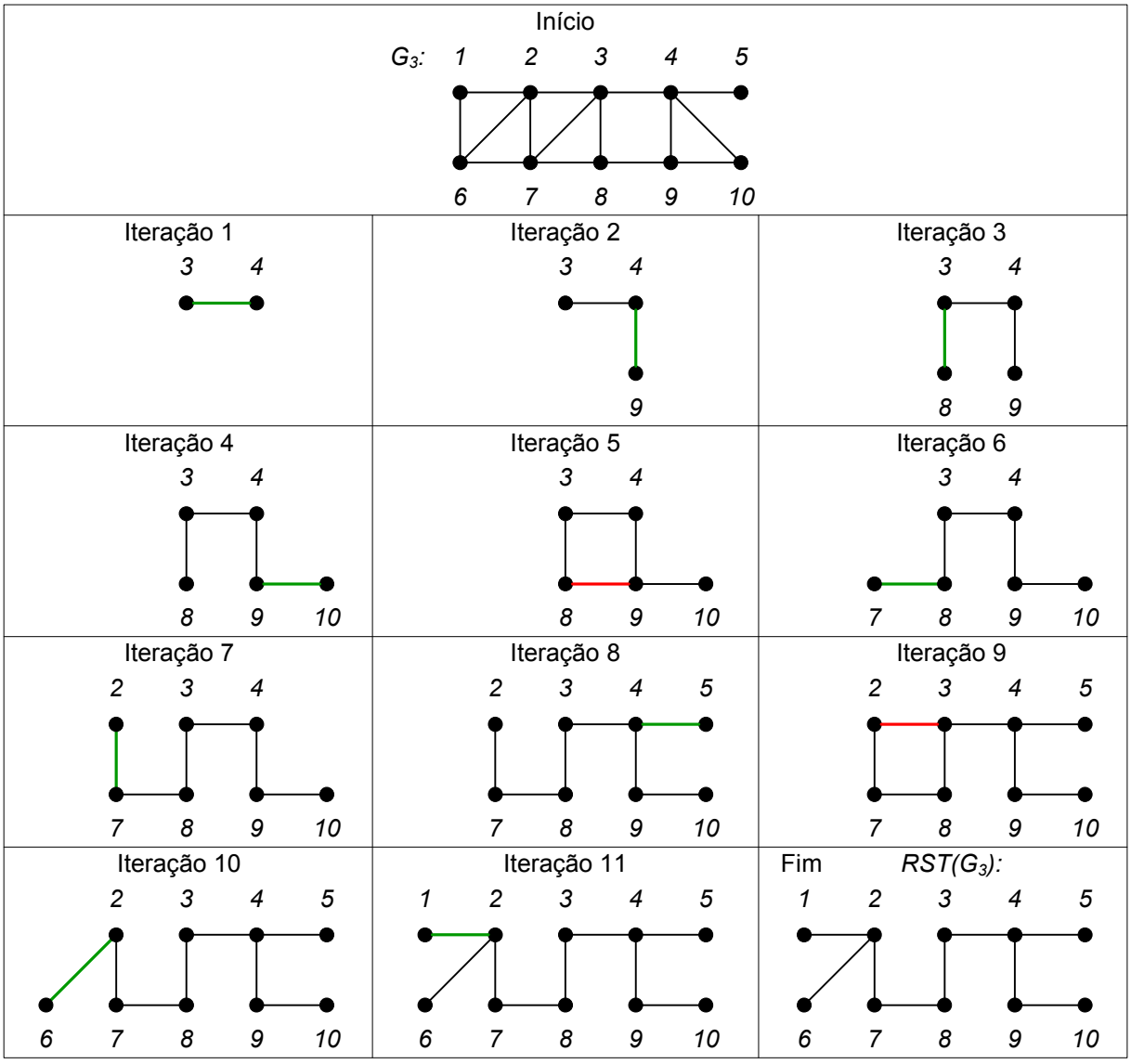

Figura 5.5: Representação gráfica dos passos executados na Tabela 5.2.

A árvore $RST\,(G_3)$ é constituída pelo conjunto de arestas:

$$F = \{\{3,4\},\{4,9\},\{3,8\},\{9,10\},\{8,7\},\{7,2\},\{4,5\},\{2,6\},\{2,1\}\}.$$

□

Note-se que, para cada par de progenitores, o método de cruzamento *One Point Crossover* gera, no máximo, dois filhos admissíveis, enquanto que o método de cruzamento *PrimRST* gera, no máximo, um filho admissível. Se o *número de torneios* for o mínimo admissível, 2, então os valores anteriores são os máximos para cada método de cruzamento de cada codificação em estudo. Por outro lado, se o *número de torneios* for superior a 2, o algoritmo cruza todos os pares de pais possíveis, maximizando a diversidade genética da população através da geração de filhos e traduzindo-se numa melhor exploração do espaço de pesquisa. Considerando t o *número de torneios*, o número de cruzamentos é dado por $\binom{t}{2} = \frac{t!}{2(t-2)!}$. Por exemplo, um algoritmo genético que convirja ao fim de 1000 iterações e realize 3 torneios por iteração, executa $\frac{3!}{2} = 3$ operações de cruzamento por iteração, totalizando 3000 operações de cruzamento. Dessas 3000 operações de cruzamento, são gerados no máximo 6000 filhos pelo método de cruzamento *One Point Crossover* e 3000 filhos pelo método de cruzamento *PrimRST*. Podemos concluir que o método de cruzamento *One Point Crossover* permite, no máximo, explorar o "dobro" do espaço de pesquisa, comparativamente com o método de cruzamento *PrimRST*.

Considerando uma população com 50 indivíduos e sabendo que em cada iteração são gerados 2 indivíduos pelo método de cruzamento *One Point Crossover*, a *amplitude geracional* (Secção 3.4) varia entre 0%, para o caso de nenhum indivíduo gerado ser adicionado à população, e 4% ($\frac{2}{50}100$), para o caso de ambos os indivíduos gerados serem adicionados à população. Para o método de cruzamento *PrimRST* é gerado 1 indivíduo em cada iteração, correspondendo a uma *amplitude geracional* que varia entre 0% e 2%.

Passo 7

O sétimo passo corresponde ao passo da mutação. A mutação é um passo facultativo, sendo aplicado apenas a alguns cromossomas. A sua aplicação é efectuada com base no parâmetro *percentagem de mutação* que indica o número de cromossomas gerados que serão mutados. Se este parâmetro for zero então os cromossomas gerados não sofrem qualquer mutação durante a execução do algoritmo genético. Neste caso o algoritmo genético perde,

gradualmente diversidade genética, convergindo rapidamente para um possível óptimo local. Se este parâmetro se aproximar de cem então os cromossomas sofrem frequentemente uma mutação durante a execução do algoritmo genético, transformando o algoritmo genético num algoritmo de pesquisa aleatória (*Random Search Algorithm*) [37].

Os cromossomas codificados por sequências de Prüfer utilizam o método de mutação *Single Point Mutation*, referido na Secção 3.6, que consiste em alterar um gene do cromossoma num locus aleatório.

A mutação para os cromossomas codificados por sequências de arestas consiste em introduzir uma aresta na árvore de suporte e remover uma aresta do ciclo formado. Esta operação de mutação corresponde à primeira estratégia de mutação apresentada na Secção 4.3.

Note-se que o parâmetro *percentagem de mutação* corresponde à percentagem de mutação da população, apresentada na Secção 3.6. A percentagem de mutação do cromossoma corresponde a um valor fixo unitário, isto é, cada cromossoma sujeito à operação de mutação tem apenas um gene modificado.

Passo 8

O oitavo passo é análogo ao terceiro passo do algoritmo genético, no entanto os indivíduos a descodificar são os indivíduos obtidos por cruzamento e/ou mutação nos sexto e sétimo passos, respectivamente.

Passo 9

Após a descodificação dos indivíduos é necessário avaliar a sua admissibilidade e calcular as suas aptidões, o que constitui o nono passo. Os métodos de cruzamento e de mutação usados não contemplam as restrições de salto, logo estes métodos podem gerar árvores de suporte que violam a restrição de salto do problema. Este passo consiste em determinar se todas as arestas das árvores de suporte obtidas pertencem ao grafo do problema, verificar se essas árvores cumprem a restrição de salto do problema e calcular o custo total de cada árvore de suporte. Note-se que se o grafo do problema for completo, é desnecessário determinar se todas as arestas das árvores de suporte obtidas pertencem ao problema.

Passo 10

O décimo passo consiste na renovação da população com indivíduos gerados por cru-

zamento e/ou mutação nos passos 6 e 7. A renovação da população processa-se em bloco (Secção 3.7), de acordo com o quarto método de substituição apresentado na Secção 3.7. Neste método os indivíduos menos aptos da população são sempre substituídos pelos novos indivíduos mesmo que a aptidão destes seja inferior. Ainda no âmbito da renovação da população é utilizado o elitismo apresentado na Secção 3.8, na qual o indivíduo mais apto da população é propagado para a nova geração.

A mutação é um mecanismo que permite manter a heterogeneidade da população. Um outro mecanismo que permite manter a heterogeneidade da população consiste em gerar novas populações durante a execução do algoritmo genético. Para não perder o trabalho efectuado pelo algoritmo genético até à iteração actual, o indivíduo mais apto é propagado para a nova população. Este mecanismo é especialmente útil em populações de pequena dimensão, que convergem rapidamente para soluções óptimas locais, permitindo uma melhor exploração do espaço de pesquisa sem grande impacto na utilização da memória do computador. O parâmetro *número de iterações para a renovação da população* permite definir o número de iterações em que uma população é mantida até ser gerada uma nova população que contém o elemento mais apto da população anterior.

Note-se que após a renovação da população, os indivíduos encontram-se ordenados por aptidão.

Passo 11

O último passo do algoritmo genético consiste em verificar o critério de paragem definido pelo parâmetro *número máximo de iterações do algoritmo genético*. O algoritmo genético pára quando o número máximo de iterações é atingindo, caso contrário passa para a iteração seguinte efectuando o quinto passo que consiste em seleccionar um novo conjunto de indivíduos para o processo de cruzamento.

5.2.1 Propriedades das Codificações Para o Problema da Árvore de Suporte de Custo Mínimo com Restrições de Salto

Na Secção 4.1 foram apresentadas as propriedades que as codificações devem cumprir. Nas Secções 4.2 e 4.3 foram referidas as propriedades respeitadas pelas codificações por sequências de Prüfer e por sequências de arestas, para o problema da *Árvore de Suporte de Custo Mínimo*, considerando o problema com grafo completo. Verificamos que a codificação

por sequências de Prüfer não respeita as propriedades da *causalidade* e da *hereditariedade*, enquanto que a codificação por sequências de arestas respeitas todas as propriedades.

Consideremos o problema da *Árvore de Suporte de Custo Mínimo com Restrições de Salto* num problema com grafo completo.

Para ambas as codificações verificamos que a propriedade da *eficiência* é garantida pelos algoritmos usados na codificação e descodificação dos cromossomas. A propriedade da *não redundância* é verificada, porque todas as soluções possuem uma única codificação. Notamos que a região de admissibilidade não é a mesma do problema da *Árvore de Suporte de Custo Mínimo*, correspondendo a um subconjunto desta constituído pelas árvores de suporte que verificam a restrição de salto. A propriedade da *completude* é verificada, porque qualquer solução para o problema tem uma codificação associada.

A codificação por sequências de Prüfer não respeita as propriedades da *causalidade* e da *hereditariedade* pelas mesmas razões mencionadas na Secção 4.2. Por outro lado, a codificação por sequências de arestas respeita ambas as propriedades anteriores, porque qualquer alteração num gene de um cromossoma com codificação por sequências de arestas, traduz-se numa alteração do factor correspondente (*causalidade*), e consequentemente a *hereditariedade* é forte.

A propriedade da *legalidade* não é verificada por ambas as codificações, porque uma permutação num elemento codificado pode originar uma solução não admissível, isto é, podemos obter uma árvore de suporte que não verifique a restrição de salto do problema. Consequentemente os métodos genéticos de cruzamento e mutação para ambas as codificações, podem gerar soluções não admissíveis.

Apesar destas codificações não respeitarem todas as propriedades, podem ser aplicadas ao problema da *Árvore de Suporte de Custo Mínimo com Restrições de Salto* recorrendo a *mecanismos adicionais* para contornar o não cumprimento da propriedade da *legalidade*.

O primeiro *mecanismo* está implementado no segundo passo, aquando da geração da população. Se a população for gerada pelo método aleatório, os indivíduos só entram na população se respeitarem a restrição de salto [26]. No entanto esta geração aleatória da população tem um tempo de processamento elevado, devido ao grande número de indivíduos que têm de ser eliminados. Se a população for gerada recorrendo à heurística *HRST* (Algoritmo 5.1), então cada indivíduo gerado respeita a restrição de salto do problema.

O segundo *mecanismo* está implementado no nono passo e consiste em incluir a verificação de admissibilidade dos cromossomas cruzados e/ou mutados, isto é, a verificação do cumprimento da restrição de salto, descrita no último paragrafo da Secção 5.1.

Estes dois mecanismos garantem a admissibilidade de todos os indivíduos da população em qualquer geração do algoritmo genético.

No Capítulo 7 é apresentado um algoritmo, baseado no método *PrimRST* (Algoritmo 5.2), para a codificação por sequências de arestas, que dispensa estes *mecanismos adicionais* para o cumprimento das restrições de salto do problema.

5.3 Módulo de Testes

No *módulo de testes* são definidos os testes a executar para cada problema. Cada teste define os valores dos parâmetros do algoritmo genético e é executado um determinado número de vezes, permitindo obter tempos de processamento mínimos, máximos, médios e totais. São também extraídos os elementos mais e menos aptos em cada teste, permitindo obter uma média e desvio padrão para um determinado número de execuções do teste. Cada teste indica o número de cruzamentos e mutações ocorridas, assim como as percentagens de sucesso ou insucesso relativamente à admissibilidade dos cromossomas obtidos por cruzamento e/ou mutação. Para finalizar a extracção de informação, a informação acima descrita é exportada para uma folha de cálculo com o formato Microsoft Excel, recorrendo à biblioteca *Java Excel API*. A informação extraída dos testes efectuados é apresentada no Capítulo 6.

Capítulo 6

Resultados Computacionais

Neste capítulo serão apresentados os resultados computacionais dos testes efectuados ao algoritmo genético implementado para o problema da *Árvore de Suporte de Custo Mínimo com Restrições de Salto*, considerando as codificações dos cromossomas por sequências de Prüfer e por sequências de arestas.

Os testes foram efectuados num computador AMD Athlon 64 2.0GHz com 2GB de memória, sobre a plataforma Java 6u18 e directamente no ambiente de programação Netbeans 6.8. A biblioteca usada para exportar a informação dos testes para uma folha de cálculo foi a *Java Excel API* 2.6.12.

Nesta dissertação foram usadas algumas das instâncias de teste usadas por Gouveia et al. [6, 17, 20, 21], nomeadamente instâncias TC, TE e TR. As instâncias TC e TE têm custos euclidianos, enquanto que as instâncias TR têm custos aleatórios. Estas instâncias constroem-se de acordo com o seguinte procedimento [6, 17]. Consideremos um inteiro n, numa grelha de dimensão 100×100, são dispostos $n+1$ pontos aleatoriamente. As instâncias TC têm os seus custos euclidianos construídos a partir das distâncias euclidianas entre um vértice raiz, no centro da grelha, e os restantes n vértices na grelha. As instâncias TE têm os seus custos euclidianos construídos a partir das distâncias euclidianas entre um vértice raiz, num extremo da grelha, e os restantes n vértices na grelha. As instâncias TR têm os seus custos definidos por valores aleatórios entre 0 e 100. Os testes efectuados consideraram instâncias TC1 com 21, 41, 61, 81, 101, 121 e 161 vértices, e instâncias TE1 e TR1 com 21, 41, 61 e 81 vértices, e a restrição de salto H com valores de 3, 4 e 5. Note-se que as instâncias TC1, TE1 e TR1 representam as primeiras instâncias de teste de um conjunto de 5 instâncias de teste para TC, TE e TR, respectivamente.

Este capítulo está organizado em três secções. Na primeira secção será comparada a performance do algoritmo genético, quando a população é gerada de forma aleatória e quando a população é gerada pela heurística *HRST* (Algoritmo 5.1), para ambas as codificações em estudo. Na segunda secção é apresentado o impacto de cada parâmetro do algoritmo genético na solução obtida. Na terceira secção são apresentadas as soluções obtidas para cada instância (TC1, TE1 e TR1), considerando a melhor configuração de teste da secção anterior, com a população gerada pela heurística *HRST* (Algoritmo 5.1).

A leitura das tabelas apresentadas neste capítulo, deve ser acompanhada do glossário apresentado após o Capítulo 8 (pg. 95) onde são indicados os significados das abreviaturas usadas.

6.1 Testes à Geração da População

Na Secção 5.2 foram referidos dois métodos para gerar a população do algoritmo genético, um método aleatório e um método heurístico (Algoritmo 5.1). Nesta secção são apresentados os *tempos de uma execução* do algoritmo genético considerando cada um destes métodos e considerando ambas as codificações de cromossomas em estudo.

A Tabela 6.1 apresenta os tempos de processamento do algoritmo genético usando o método de geração aleatória da população, enquanto que a Tabela 6.2 apresenta os tempos de processamento do algoritmo genético usando a Heurística *HRST* para geração da população. Ambas as tabelas apresentam os tempos de processamento para a codificação de cromossomas por sequências de Prüfer e por sequências de arestas. As tabelas são constituídas por oito colunas. A primeira identifica a instância testada, a segunda indica a ordem da instância (N), a terceira indica o valor do salto (H), a quarta indica o valor do custo óptimo (OPT) associado a essa instância e valor de salto. A quinta e sexta colunas indicam, respectivamente, o custo do elemento mais apto (CEMA) obtido através do algoritmo genético e o tempo de processamento do algoritmo (T) para a codificação por sequências de Prüfer. Analogamente para a sétima e oitava colunas e codificação por sequências de arestas.

As instâncias usadas no teste foram as TC1 com $N = 20$ e $N = 40$, considerando a restrição de salto $H = 3, 4, 5$. Pela tabela concluímos que apenas é aceitável a geração aleatória da população para instâncias pequenas e com a restrição de salto igual ou superior a 4. Com a diminuição do valor da restrição de salto para 3, torna-se mais difícil gerar

				Codificação por Sequências de Prüfer		Codificação por Sequências de Arestas	
	N	H	OPT	CEMA	T	CEMA	T
Geração Aleatória da População							
TC1	20	3	340	398	1053	403	1085
	20	4	318	399	17	330	16
	20	5	312	424	4	322	4
TC1	40	3	609	-	>24h	-	>24h
	40	4	548	-	>24h	-	>24h
	40	5	522	1079	3453	770	3628

Tabela 6.1: Representação dos tempos de processamento do algoritmo genético usando o método de geração aleatória da população.

				Codificação por Sequências de Prüfer		Codificação por Sequências de Arestas	
	N	H	OPT	CEMA	T	CEMA	T
Geração da População usando a Heurística *HRST*							
TC1	20	3	340	352	8	340	8
	20	4	318	328	8	318	8
	20	5	312	316	8	312	8
TC1	40	3	609	745	27	772	24
	40	4	548	713	27	696	25
	40	5	522	660	27	638	25

Tabela 6.2: Representação dos tempos de processamento do algoritmo genético usando a heurística *HRST* para geração da população.

aleatoriamente árvores de suporte que respeitem esta restrição. O tempo de processamento para a instância TC1 com $N = 40$, usando a geração aleatória da população, é superior a 24h para valores de salto $H = 3, 4$.

Todos os testes realizados nas próximas secções consideram, apenas, a geração da população através da heurística *HRST*.

6.2 Testes aos Parâmetros do Algoritmo Genético

Nesta secção são realizados testes às instâncias TC1, TE1 e TR1 com $N = 20$, com a finalidade de analisar o impacto de cada parâmetro do algoritmo genético, descrito na Secção 5.2, na solução obtida.

A Tabela 6.3 apresenta doze configurações de teste, isto é, apresenta doze testes com valores distintos para um ou mais parâmetros do algoritmo genético. A tabela é constituída por sete colunas. Na primeira coluna está identificado o número da configuração do teste (NCT), na segunda está representado o número máximo de iterações (NMI) do algoritmo genético (parâmetro do critério de paragem do algoritmo genético), na terceira coluna temos a dimensão da população (DP), na quarta a dimensão de cada torneio (DT), na quinta o número de torneios por cada iteração (NT) do algoritmo genético, na sexta coluna está representada a percentagem de mutação (PM) para os indivíduos cruzados, e na sétima coluna temos o número de iterações para renovação da população (NIRP).

NCT	NMI	DP	DT	NT	PM	NIRP
1	10000	50	3	2	5	1000
2	**1000**	50	3	2	5	1000
3	10000	**500**	3	2	5	1000
4	10000	50	**10**	2	5	1000
5	10000	50	**25**	2	5	1000
6	10000	50	3	**3**	5	1000
7	10000	50	3	2	**20**	1000
8	10000	50	3	2	**95**	1000
9	10000	50	3	2	5	**200**
10	10000	50	3	2	5	**50**
11	10000	50	**10**	2	5	**200**
12	10000	50	**10**	2	**20**	**200**

Tabela 6.3: Representação dos valores dos parâmetros associados a cada teste.

O *teste 1* será considerado o teste base. Os restantes testes são variações do *teste 1* em um ou mais parâmetros assinalados a negrito. A leitura do *teste 1* processa-se do seguinte modo: "para a configuração de *teste 1*, o algoritmo genético pára após 10000 iterações, a população é constituída por 50 indivíduos, em cada torneio concorrem 3 indivíduos e realizam-se 2 torneios, 5% dos indivíduos cruzados são mutados, e a população é renovada a cada 1000 iterações (sendo preservado o indivíduo mais apto)". Analogamente para os

restantes testes com as diferenças assinaladas a negrito.

Cada teste foi repetido 100 vezes para cada uma das codificações em estudo. Os resultados para a instância TC1, para as codificações de cromossomas por sequências de Prüfer e por sequências de arestas, são apresentados nas Tabelas 6.4 e 6.5, respectivamente. Devido à extensão destas tabelas, os resultados para as instâncias TE1 e TR1 são apresentados nas Tabelas 1 a 4, no Apêndice A.

Estas tabelas são compostas por doze colunas. A primeira coluna indica a configuração do teste (NCT), a segunda indica o custo do melhor indivíduo (CEMA) obtido nas 100 repetições, a terceira apresenta a média dos custos (MCE) para as 100 repetições, a quarta apresenta o desvio padrão (DPCE), a quinta indica o número de cruzamentos efectuados numa repetição (NTC), a sexta e sétima colunas indicam as percentagens de sucesso (PMCS) e insucesso (PMCI) para os elementos cruzados, isto é, se esses elementos formam árvores de suporte que respeitam a restrição de salto. A oitava coluna apresenta o número médio de mutações (NMM) ocorridas nas 100 repetições, a nona e a décima apresentam as taxas de sucesso (PMMS) e insucesso (PMMI) para as mutações, de forma análoga às sexta e sétima colunas. A décima primeira indica o tempo médio (TM), em segundos, de cada repetição e a última indica o tempo total (TT), em minutos, para as 100 repetições. A negrito estão representados os melhores valores para o custo do melhor indivíduo (CEMA) e para a média dos custos (MCE). Os testes que geraram elementos óptimos são representados a verde.

Analisando as Tabelas 6.4 e 6.5 em simultâneo, verificamos que o algoritmo genético com codificação por sequências de Prüfer não conseguiu encontrar o valor óptimo com nenhum teste, enquanto que o algoritmo genético com codificação por sequências de arestas conseguiu encontrar o valor óptimo para as três restrições de salto do problema em pelo menos um teste. A dispersão dos resultados é superior com a codificação de Prüfer em relação à codificação por arestas. Em todos os testes a percentagem de cruzamentos com sucesso é superior com a codificação de Prüfer em relação à codificação por arestas, no entanto verifica-se o inverso em relação à percentagem de mutação com sucesso. Os tempos totais são praticamente idênticos para ambas as codificações.

O *teste 1* apresenta bons resultados em todos os casos, estando próximo do valor óptimo. Os resultados do *teste 2* permite concluir que o número de iterações considerado é baixo para explorar o espaço de pesquisa, apresentando médias e dispersões de custos

Codificação por Sequências de Prüfer											
NCT	CEMA	MCE	DPCE	NTC	PMCS	PMCI	NMM	PMMS	PMMI	TM	TT
TC1, N=20, H=3, OPT=340											
1	359	398.3	19.8	20000	98.4	1.6	981.6	50.9	49.1	2.3	3.8
2	419	500.5	32.1	2000	98.6	1.4	98.0	70.1	29.9	0.0	0.0
3	447	493.5	19.3	20000	86.9	13.1	871.2	84.6	15.4	4.1	6.9
4	365	402.7	22.5	20000	99.6	0.4	1001.8	45.9	54.1	2.3	3.8
5	361	408.7	24.6	20000	99.8	0.2	1001.6	46.3	53.7	2.6	4.4
6	352	391.7	20.2	60000	99.0	1.0	2972.6	44.3	55.7	8.0	13.3
7	358	392.0	19.4	20000	98.1	1.9	3922.7	47.1	52.9	3.0	5.0
8	**347**	**371.4**	12.5	20000	91.7	8.3	18331.0	54.8	45.2	4.6	7.7
9	356	399.5	21.9	20000	92.4	7.6	925.9	66.5	33.5	3.0	5.0
10	431	480.4	23.0	20000	82.9	17.1	822.7	81.0	19.0	5.0	8.4
11	357	401.1	22.6	20000	97.9	2.1	977.0	49.3	50.8	3.0	5.0
12	359	393.8	22.4	20000	97.6	2.4	3902.4	44.5	55.6	3.1	5.1
TC1, N=20, H=4, OPT=318											
1	332	377.1	21.9	20000	98.9	1.1	989.0	67.1	32.9	2.3	3.8
2	423	486.1	32.2	2000	98.9	1.1	98.0	84.5	15.5	0.0	0.0
3	411	481.6	19.6	20000	90.7	9.3	905.9	90.8	9.2	4.4	7.3
4	338	384.8	27.9	20000	99.7	0.3	996.6	62.3	37.7	2.4	4.0
5	342	389.9	28.8	20000	99.9	0.1	992.1	61.5	38.5	2.8	4.7
6	**328**	371.8	25.8	60000	99.3	0.7	2975.9	58.5	41.6	8.0	13.3
7	335	370.1	23.1	20000	98.8	1.3	3953.8	58.7	41.3	3.0	5.0
8	332	**352.7**	14.1	20000	95.1	4.9	19021.1	65.5	34.5	4.9	8.2
9	340	383.2	20.7	20000	94.8	5.2	948.6	78.4	21.6	3.0	5.0
10	407	470.8	22.9	20000	87.8	12.2	879.1	88.4	11.6	5.0	8.3
11	332	380.0	23.9	20000	98.7	1.3	985.9	65.9	34.1	3.0	5.0
12	335	372.8	24.6	20000	98.4	1.6	3938.0	57.9	42.1	3.1	5.2
TC1, N=20, H=5, OPT=312											
1	324	373.7	28.2	20000	99.2	0.8	993.5	78.5	21.5	2.4	4.0
2	419	488.8	30.6	2000	99.2	0.8	98.5	90.2	9.8	0.0	0.0
3	434	485.2	20.2	20000	92.5	7.5	916.6	93.6	6.4	4.4	7.3
4	324	384.4	28.2	20000	99.8	0.2	1006.5	73.7	26.3	2.4	4.1
5	332	383.5	29.0	20000	99.9	0.1	998.7	74.4	25.6	2.8	4.7
6	**316**	368.3	26.4	60000	99.5	0.5	2988.2	72.8	27.2	8.0	13.3
7	322	364.9	21.8	20000	99.1	0.9	3971.6	73.5	26.5	3.0	5.0
8	318	**346.7**	16.2	20000	97.1	2.9	19411.1	76.7	23.3	5.0	8.3
9	322	374.0	21.5	20000	96.1	3.9	959.2	86.0	14.0	3.0	5.0
10	398	465.7	25.6	20000	90.7	9.3	908.6	92.0	8.0	5.0	8.3
11	330	375.9	25.7	20000	99.0	1.0	992.2	77.2	22.8	3.0	5.0
12	324	369.6	22.7	20000	98.8	1.2	3943.3	72.5	27.5	3.2	5.3

Tabela 6.4: Representação dos resultados dos testes 1 a 12 para TC1 com $N = 20$ e codificação de cromossomas por sequências de Prüfer.

Codificação por Sequências de Arestas											
NCT	CEMA	MCE	DPCE	NTC	PMCS	PMCI	NMM	PMMS	PMMI	TM	TT
TC1, N=20, H=3, OPT=340											
1	354	378.8	15.0	10000	79.79	20.2	495.4	69.1	30.9	3.0	5.0
2	417	493.0	31.6	1000	86.12	13.9	48.9	79.9	20.1	0.0	0.0
3	497	541.7	14.0	10000	71.49	28.5	498.1	68.6	31.4	4.0	6.7
4	382	439.4	23.5	10000	93.73	6.3	465.5	70.0	30.0	2.0	3.3
5	436	489.5	22.7	10000	96.50	3.5	445.5	73.4	26.6	2.0	3.4
6	**340**	**361.3**	7.9	30000	83.03	17.0	1474.6	71.0	29.0	8.0	13.3
7	349	383.2	12.5	10000	73.56	26.4	1979.5	65.6	34.4	3.0	5.0
8	391	426.2	13.6	10000	55.88	44.1	10000.0	55.9	44.1	4.0	6.7
9	439	483.2	14.8	10000	76.46	23.5	498.6	73.6	26.4	3.0	5.0
10	493	529.9	18.0	10000	62.41	37.6	499.2	59.2	40.8	4.3	7.1
11	351	390.1	16.5	10000	80.25	19.8	490.1	65.5	34.5	3.0	5.0
12	363	396.1	15.1	10000	72.53	27.5	1941.7	61.8	38.2	3.0	5.0
TC1, N=20, H=4, OPT=318											
1	328	346.5	11.7	10000	85.8	14.2	496.0	77.4	22.6	3.0	5.0
2	392	467.6	29.2	1000	92.4	7.6	49.5	89.1	10.9	0.0	0.0
3	471	524.6	15.1	10000	84.8	15.2	495.8	83.5	16.5	4.2	6.9
4	346	411.0	24.1	10000	95.7	4.3	467.2	78.7	21.3	2.0	3.4
5	386	460.8	24.0	10000	98.0	2.0	446.1	84.0	16.0	2.1	3.4
6	**318**	**332.1**	7.9	30000	87.1	12.9	1479.7	76.5	23.5	8.0	13.3
7	326	350.8	10.1	10000	81.8	18.3	1969.6	75.7	24.3	3.0	5.0
8	368	392.3	11.6	10000	69.5	30.5	10000.0	69.5	30.5	4.0	6.7
9	394	451.0	14.3	10000	86.6	13.4	496.0	85.0	15.0	3.0	5.0
10	473	512.3	16.9	10000	79.5	20.5	496.9	78.0	22.0	4.6	7.7
11	**318**	356.5	12.5	10000	87.0	13.0	492.2	75.8	24.2	3.0	5.0
12	322	359.6	14.7	10000	80.8	19.2	1936.7	72.4	27.6	3.0	5.0
TC1, N=20, H=5, OPT=312											
1	314	329.3	10.8	10000	89.5	10.5	493.2	83.0	17.0	3.0	5.0
2	356	445.9	31.6	1000	94.9	5.2	50.4	93.1	6.9	0.0	0.0
3	452	513.3	14.9	10000	91.9	8.2	498.4	91.1	8.9	4.5	7.5
4	337	390.9	21.2	10000	97.3	2.7	462.2	85.9	14.1	2.0	3.4
5	385	450.3	32.1	10000	98.9	1.2	454.4	91.4	8.6	2.0	3.4
6	**312**	**320.0**	6.2	30000	89.5	10.5	1485.0	80.9	19.1	8.0	13.3
7	314	333.0	10.7	10000	85.9	14.1	1969.9	81.4	18.6	3.0	5.0
8	344	373.4	12.9	10000	79.9	20.1	10000.0	79.9	20.1	4.0	6.7
9	383	433.5	16.1	10000	92.3	7.7	499.8	91.7	8.3	3.0	5.0
10	443	501.7	18.8	10000	88.7	11.3	495.0	87.9	12.1	4.8	8.1
11	316	337.2	10.9	10000	90.0	10.0	488.2	80.9	19.1	3.0	5.0
12	316	343.4	11.5	10000	85.5	14.5	1945.4	78.9	21.1	3.0	5.0

Tabela 6.5: Representação dos resultados dos testes 1 a 12 para TC1 com $N = 20$ e codificação de cromossomas por sequências de arestas.

elevadas. O aumento da dimensão da população, considerado no *teste 3*, não favorece os resultados obtidos em nenhum dos casos. Por outro, lado o aumento da dimensão dos torneios nos *testes 4 e 5* favorece os resultados obtidos, com os melhores valores obtidos com uma dimensão intermédia (*teste 4*). No *teste 6* o aumento do número de cruzamentos favorece a exploração do espaço de pesquisa, obtendo os melhores resultados para a codificação por arestas. Por outro lado este teste tem um impacto negativo a nível do tempo de processamento, sendo o teste mais moroso de entre os 12 testes. Nos *testes 7 e 8* aumentamos a percentagem de mutação do algoritmo genético, o primeiro em 20% e o segundo em 95%, aproximando-o de um algoritmo de pesquisa aleatória. O *teste 7* obtém melhores resultados para a codificação por arestas, enquanto que o *teste 8* obtém melhores resultados para a codificação de Prüfer. Relembremos que o método de cruzamento *One Point Crossover* tem tendência a propagar o primeiro ou último elemento do cromossoma, com a consequência de ignorar parte do espaço de pesquisa. Com a elevada percentagem de mutação do *teste 8*, esses elementos vão ser mutados permitindo explorar esse espaço ignorado. O método de cruzamento *PrimRST* não tem este problema e quando associado a uma percentagem de mutação elevada, tende a gerar indivíduos com aptidões inferiores. Nos *testes 9 e 10* diversificamos a população, evitando convergir para óptimos locais. No entanto verificamos que se o valor para a renovação for muito baixo (*teste 10*), o algoritmo genético não consegue explorar completamente a população actual até esta ser renovada, traduzindo-se numa má exploração do espaço de pesquisa. Por fim, nos *testes 11 e 12* juntamos dois ou três parâmetros do algoritmo genético, que obtiveram bons resultados nos teste anteriores. Verificamos que a par do *teste 6*, são dos melhores testes, tendo ainda a vantagem a nível do tempo de processamento em relação ao *teste 6*.

Verificamos que obtemos bons resultados quando a dimensão da população é de 50 indivíduos. O melhor valor para a dimensão dos torneios oscila entre 3 (*teste 1*) e 10 (*teste 4*). O melhor valor para a percentagem de mutação oscila entre 5% (*teste 1*) e 20% (*teste 7*). A renovação da população deve oscilar entre as 200 iterações (*teste 9*) e as 1000 iterações (*teste 1*). O tempo de uma execução para estes testes oscila entre os 3 e os 5 segundos para ambas as codificações. Verificamos ainda que os melhores resultados são produzidos, geralmente, pela configuração do *teste 6*, que aumenta o número de indivíduos cruzados por iteração, permitindo uma melhor exploração do espaço de pesquisa.

Para a instância TE1 com $N = 20$ (Tabelas 1 e 2, Apêndice A), verificamos para ambas

as codificações, que o algoritmo genético não gerou indivíduos óptimos, sendo as médias e dispersões obtidas elevadas. Tal como para a instância TC1 com $N = 20$, a percentagem de cruzamentos com sucesso é superior para a codificação de Prüfer. Contrariamente à instância TC1 com $N = 20$, a percentagem de mutação com sucesso é inferior para a codificação por arestas. Os tempos de processamento são ligeiramente superiores em relação à instância TC1 com $N = 20$. O *teste 6* revelou-se o melhor teste para a codificação por sequências de arestas, enquanto que o *teste 8* (para $H = 3, 4$) e o *teste 12* (para $H = 5$) revelaram-se os melhores para a codificação por sequências de Prüfer. Novamente a percentagem de mutação elevada a auxiliar o método de cruzamento *One Point Crossover* para a codificação de Prüfer.

Para a instância TR1 com $N = 20$ (Tabelas 3 e 4, Apêndice A), verificamos que o algoritmo genético não gerou indivíduos óptimos para a codificação de Prüfer, contudo gerou indivíduos óptimos para a codificação por arestas. Verificamos que a percentagem de cruzamentos com sucesso é superior para a codificação de Prüfer, no entanto a percentagem de mutação com sucesso é idêntica para ambas as codificações. Os tempos de processamento são semelhantes aos da instância TE1 com $N = 20$. O *teste 6* revelou-se o melhor teste para a codificação por sequências de Prüfer quando a restrição de salto tem valor 3. Para valores de salto de 4 e 5, os testes que aumentam a percentagem da mutação (*testes 7 e 8*) revelaram-se melhores. Em relação à codificação por sequências de arestas, o *teste 6* revelou-se o melhor. No entanto, para o valor de salto 5, os *testes 1, 7, 11 e 12* também geraram indivíduos óptimos.

Concluímos que os tempos totais de processamento das instâncias, para ambas as codificações, diferem ligeiramente em poucos minutos (diferenças máximas de 2 minutos para 100 execuções), podendo ser considerados idênticos. Note-se que no mesmo período de tempo, o algoritmo genético com codificação por sequências de Prüfer consegue obter duas vezes mais indivíduos, pela operação de cruzamento, em relação ao algoritmo genético com codificação por sequências de arestas, fruto da simplicidade do método de cruzamento *One Point Crossover* que consegue absorver o tempo perdido com as codificações e descodificações dos cromossomas. Apesar do algoritmo genético com codificação por sequências de arestas não necessitar codificar e descodificar os cromossomas, a junção dos cromossomas progenitores num grafo auxiliar e o método de cruzamento *PrimRST*, revelam-se mais

morosos.

O algoritmo genético com codificação por sequências de Prüfer beneficia da elevada percentagem de mutação que é usada para evitar os problemas associados à propagação do primeiro ou último elemento de cada cromossoma. Esta elevada percentagem de mutação pode acabar por transformar o algoritmo genético com codificação por sequências de Prüfer, num algoritmo de pesquisa aleatória (*teste 8*).

O método de cruzamento *One Point Crossover* gera indivíduos admissíveis com maior probabilidade, em relação ao método *PrimRST*. A vantagem no uso da operação de mutação torna-se ambígua, pois para a instância TC1 com $N = 20$, a codificação por arestas apresenta maiores percentagens de indivíduos mutados com sucesso, para a instância TE1 com $N = 20$, verificamos o contrário, e para a instância TR1 com $N = 20$ as percentagens são idênticas.

O algoritmo genético, com codificação por sequências de arestas, conseguiu gerar os elementos óptimos para as instâncias TC1 e TR1 com $N = 20$.

6.3 Performance da Melhor Configuração Para Outras Instâncias

Nesta secção serão apresentados os resultados para a melhor configuração de teste, apresentada na Secção 6.2.

Na secção anterior foram apresentados doze testes para as instâncias TC1, TE1 e TR1 com $N = 20$. Estes doze testes foram executados 100 vezes para cada instância, considerando valores de salto $H = 3, 4, 5$ e ambas as codificações em estudo. Um dos testes que mais se destacou, pelos bons resultados obtidos, foi o da configuração de *teste 6*. Esta configuração será aplicada às instâncias TC1 com $N = 40, 60, 80, 100, 120, 160$, e às instâncias TE1 e TR1 com $N = 40, 60, 80$, para determinar a performance do algoritmo genético e as soluções obtidas. Tal como na secção anterior, será executado 100 vezes para cada instância.

Os resultados obtidos para as instâncias TC1 com $N = 40, 60, 80, 100, 120, 160$ são apresentados nas Tabelas 6.6 e 6.7. Os resultados obtidos para as instâncias TE1 e TR1, com $N = 40, 60, 80$, são apresentados nas Tabela 5 e 6 do Anexo A. Para que estas tabelas fiquem completas, os resultados das instâncias TC1, TE1 e TR1 com $N = 20$ são também

apresentados nas respectivas tabelas.

As tabelas são constituídas por treze colunas. A primeira coluna indica a ordem da instância (N), a segunda indica o valor do salto (H), a terceira corresponde ao valor de custo óptimo (OPT), a quarta indica o custo de elemento mais apto (CEMA) obtido numa das 100 execuções, a quinta contém o valor do *gap* (Gap, ver Glossário pg. 95), isto é, uma medida que indica a percentagem da diferença entre o melhor valor obtido (CEMA) e o valor óptimo (OPT). A sexta e sétima colunas correspondem à média (MCE) e dispersão (DPCE) dos custos obtidos nas 100 execuções, a oitava e a nona colunas correspondem à percentagem de cruzamentos com sucesso (PMCS) e sem sucesso (PMCI), a décima e a décima primeira colunas correspondem à percentagem de mutações com sucesso (PMMS) e sem sucesso (PMMI), a décima segunda indica o tempo médio (TM), em segundos, de uma execução e a décima terceira indica o tempo total (TT), em minutos, para as 100 execuções.

Para as instâncias TC1 com $N = 20, 40, 60, 80, 100, 120, 160$ (Tabelas 6.6 e 6.7), verificamos a tendência referida na secção anterior, em que a percentagem de cruzamentos com sucesso é superior para a codificação por sequências de Prüfer, e a percentagem de mutação com sucesso é superior para a codificação por sequências de arestas. O valor do gap indica que a partir da instância TC1 com $N = 60$, a codificação de Prüfer, obteve melhores valores de custo para os indivíduos mais aptos. Constatamos que o aumento do número de vértices produz um grande impacto no tempo de processamento, que é superior para a codificação de Prüfer.

Para as instâncias TE1 com $N = 20, 40, 60, 80$ (Tabelas 5 e 6, Apêndice A), verificamos que, a partir da instância TE1 com $N = 40$, a codificação de Prüfer obtém os melhores resultados de custo dos indivíduos mais aptos. Verificamos ainda, que a codificação por sequências de arestas tem percentagens de cruzamentos e de mutação inferiores à codificação por sequências de Prüfer, explicando o seu fraco desempenho. A nível dos tempos de processamento, ambas as codificações apresentam tempos semelhantes.

Para as instâncias TR1 com $N = 20, 40, 60, 80$ (Tabelas 7 e 8, Apêndice A), verificamos que as conclusões são semelhantes às das instâncias TE. No entanto existem duas excepções, a percentagem de mutação com sucesso e os tempos de processamento são favoráveis à codificação por sequências de arestas.

Codificação por Sequências de Prüfer												
N	H	OPT	CEMA	Gap	MCE	DPCE	PMCS	PMCI	PMMS	PMMI	TM	TT
20	3	340	352	3.5	391.7	20.2	99.0	1.0	44.3	55.7	8.0	13.3
	4	318	328	3.1	371.8	25.8	99.3	0.7	58.5	41.6	8.0	13.3
	5	312	316	1.3	368.3	26.4	99.5	0.5	72.8	27.2	8.0	13.3
40	3	609	745	22.3	833.3	48.5	99.0	1.1	44.8	55.2	26.6	44.4
	4	548	713	30.1	793.3	44.3	99.3	0.7	58.2	41.8	27.0	45.0
	5	522	660	26.4	763.2	45.7	99.5	0.5	69.9	30.1	27.2	45.3
60	3	866	1195	38.0	1358.0	69.9	98.9	1.1	46.0	54.1	54.5	90.9
	4	781	1046	33.9	1278.2	74.9	99.3	0.7	60.4	39.6	55.1	91.8
	5	734	1084	47.7	1229.3	73.0	99.6	0.4	71.6	28.4	54.7	91.2
80	3	1072	1645	53.5	1830.7	87.9	98.8	1.2	47.8	52.2	97.2	161.9
	4	981	1547	57.7	1724.0	77.0	99.3	0.7	61.0	39.0	98.2	163.7
	5	922	1471	59.5	1650.7	74.7	99.5	0.5	72.3	27.7	98.2	163.7
100	3	1259	2168	72.2	2344.5	79.8	98.7	1.3	50.4	49.6	158.1	263.4
	4	1166	1980	69.8	2190.9	92.3	99.2	0.8	63.8	36.2	161.0	268.3
	5	1104	1892	71.4	2141.0	101.2	99.5	0.5	76.2	23.8	162.6	271.0
120	3	1059	2316	118.7	2598.1	107.2	98.6	1.4	51.1	48.9	239.1	398.5
	4	926	2179	135.3	2435.2	105.5	99.2	0.8	65.4	34.6	242.8	404.6
	5	853	2070	142.7	2334.1	93.1	99.5	0.5	75.6	24.4	247.7	412.9
160	3	1357	3942	190.5	4632.2	350.3	91.2	8.8	40.5	59.5	471.0	785.0
	4	1133	3691	225.8	4295.1	311.1	93.8	6.2	51.2	48.8	460.0	766.7
	5	1039	3328	220.3	3998.5	224.6	96.0	4.0	59.3	40.7	480.7	801.1

Tabela 6.6: Teste 6 para as instâncias TC1 com $N = 20, 40, 60, 80, 100, 120, 160$ e codificação de cromossomas por sequências de Prüfer.

Verificamos na secção anterior, uma tendência na qual a codificação de Prüfer apresenta percentagens de cruzamentos com sucesso superiores à codificação de arestas, uma tendência que continua a verificar-se nesta secção. A codificação de Prüfer apresentou melhores resultados, devido ao número de cruzamentos efectuado, 60000 contra os 30000 da codificação por arestas. Este elevado número de cruzamentos, permitiu uma melhor exploração do espaço de pesquisa, obtendo indivíduos com custos inferiores aos indivíduos obtidos pela codificação por arestas.

A codificação de cromossomas por sequências de arestas, juntamente com o método de cruzamento *PrimRST* e a primeira estratégia de mutação, permitem obter melhores soluções para problemas com 21 vértices, relativamente à codificação de cromossomas por sequências de Prüfer. Quando o número de vértices dos problemas é superior a 21, a codificação de cromossomas por sequências de Prüfer, juntamente com o método de cruzamento

Codificação por Sequências de Arestas												
N	H	OPT	CEMA	Gap	MCE	DPCE	PMCS	PMCI	PMMS	PMMI	TM	TT
20	3	340	340	0.0	361.3	7.9	83.0	17.0	71.0	29.0	8.0	13.3
	4	318	318	0.0	332.1	7.9	87.1	12.9	76.5	23.5	8.0	13.3
	5	312	312	0.0	320.0	6.2	89.5	10.5	80.9	19.1	8.0	13.3
40	3	609	772	26.8	869.0	38.4	82.7	17.3	75.8	24.3	24.4	40.7
	4	548	696	27.0	763.5	32.5	88.4	11.6	83.6	16.4	24.8	41.4
	5	522	638	22.2	699.8	30.8	92.1	7.9	89.5	10.5	25.2	42.0
60	3	866	1422	64.2	1552.6	54.4	83.2	16.9	76.8	23.2	45.7	76.2
	4	781	1293	65.6	1433.8	54.1	90.5	9.5	87.2	12.8	46.1	76.9
	5	734	1245	69.6	1352.3	47.9	94.9	5.1	93.3	6.7	46.6	77.6
80	3	1072	2084	94.4	2242.4	61.4	85.7	14.3	80.7	19.3	73.8	123.0
	4	981	1909	94.6	2115.8	62.9	92.9	7.1	90.7	9.3	75.8	126.3
	5	922	1888	104.8	2047.3	63.1	96.5	3.5	95.9	4.1	76.3	127.2
100	3	1259	2715	115.6	2960.7	75.5	88.1	11.9	83.9	16.1	111.1	185.2
	4	1166	2606	123.5	2821.1	72.3	94.6	5.4	93.0	7.0	113.2	188.7
	5	1104	2546	130.6	2743.7	76.1	97.8	2.2	97.3	2.7	113.6	189.3
120	3	1059	3226	204.6	3398.5	71.2	88.9	11.1	85.2	14.8	150.6	250.9
	4	926	3098	234.6	3255.5	71.4	94.7	5.3	93.6	6.4	147.0	245.0
	5	853	2970	248.2	3193.5	83.2	97.7	2.3	97.4	2.6	147.3	245.5
160	3	1357	5497	305.1	6267.3	205.3	55.9	44.1	53.8	46.2	290.0	483.3
	4	1133	4264	276.3	5522.5	473.1	71.8	28.2	70.1	29.9	291.2	485.3
	5	1039	3842	269.8	5198.3	409.4	82.1	17.9	81.3	18.7	294.7	491.2

Tabela 6.7: Teste 6 para as instâncias TC1 com $N = 20, 40, 60, 80, 100, 120, 160$ e codificação de cromossomas por sequências de arestas.

One Point Crossover e o método de mutação *Single Point Mutation*, exploram melhor o espaço de pesquisa e, consequentemente, permitem obter melhores soluções relativamente à codificação de cromossomas por sequências de arestas. No entanto, para as instâncias maiores, o tempo de processamento para a codificação de cromossomas por sequências de Prüfer aproxima-se do dobro em relação ao tempo de processamento para a codificação de cromossomas por sequências de arestas.

Resumindo, a codificação de cromossomas por sequências de arestas deve ser usada em instâncias pequenas, ou em instâncias grandes quando o tempo de processamento tem maior peso relativamente à qualidade da solução obtida. A codificação de cromossomas por sequências de Prüfer deve ser usada em instâncias grandes quando a qualidade da solução é mais importante relativamente ao tempo de processamento.

Capítulo 7

Algoritmo PrimHMRST

Neste capítulo apresentamos o método *PrimHMRST* que é usado para gerar e/ou cruzar indivíduos da população admissíveis para o problema da *Árvore de Suporte de Custo Mínimo com Restrições de Salto*.

O método de cruzamento *One Point Crossover* e o método de mutação *Single Point Mutation*, aplicados à codificação por sequências de Prüfer, não geram soluções que à partida respeitam a restrição de salto do problema, logo nem todos os filhos gerados por estes métodos respeitam a restrição de salto. Analogamente para os métodos de cruzamento (*PrimRST*) e de mutação aplicados à codificação por sequências de arestas.

Na Secção 5.2.1 foram referidos dois mecanismos que permitem a utilização de ambas as codificações para o problema da *Árvore de Suporte de Custo Mínimo com Restrições de Salto*. Estes mecanismos permitem considerar apenas os indivíduos que respeitam a restrição de salto do problema obtidos na geração da população ou no processo de cruzamento/mutação/renovação da população, eliminando os restantes indivíduos. No entanto, estes mecanismos têm algum impacto na eficiência do algoritmo genético, devido ao tempo de processamento com indivíduos que depois não são admissíveis. Para contornar esta situação, podemos adicionar as restrições de salto do problema aos operadores genéticos de cruzamento e mutação, de modo a obter sempre um indivíduo admissível em cada uma dessas operações.

Métodos genéticos de cruzamento e mutação que respeitam as restrições de salto do problema são difíceis de implementar nestas codificações. No entanto durante a implementação em Java do método de cruzamento *PrimRST* para a codificação por sequências de arestas, constatou-se a possibilidade de adicionar as restrições de salto a esse método de

Algoritmo 7.1: Método de cruzamento e mutação *PrimHMRST*, respeitando o valor de salto H da *Árvore de Suporte de Custo Mínimo com Restrições de Salto*.

Entrada: $G = (V, E) \rightarrow$ grafo auxiliar (conexo)
$r \rightarrow$ vértice raiz da árvore de suporte T
$H \rightarrow$ número máximo de saltos permitido
$B \rightarrow$ conjunto com arestas disponíveis para a mutação
Saída: $T = (C, F) \rightarrow$ árvore de suporte
Variáveis: $F \rightarrow$ conjunto das arestas da árvore de suporte T
$C \rightarrow$ conjunto dos vértices da árvore de suporte T
$A \rightarrow$ conjunto das arestas temporárias
$s, u, v, w \rightarrow$ vértices temporários
$h \rightarrow$ salto temporário

```
1  F ← {}
2  s ← r //a raiz é o vértice inicial
3  C ← {s}
4  A ← {{s, v, 1} : {s, v} ∈ E, v ∈ V} //a terceira posição indica o salto
   associado a estas arestas {s, v}
5  enquanto C ≠ V fazer
6      se A ∩ B ≠ {} então //se existe uma aresta em ambas os conjuntos,
       essa aresta tem prioridade de escolha para a realização da mutação
7          escolher aleatoriamente uma aresta {u, v, h} ∈ A tal que {u, v} ∈ B
8          B ← B \ {u, v}
9      senão //caso não haja uma aresta prioritária
10         escolher aleatoriamente uma aresta {u, v, h} ∈ A, com u ∈ C
11     A ← A \ {u, v, h}
12     se v ∉ C então //colocar a aresta {u, v} na árvore de suporte
13         F ← F ∪ {u, v}
14         C ← C ∪ v
15         A ← A ∪ {{v, w, h + 1} : {v, w} ∈ E, w ∉ C, h + 1 ≤ H} //adiciona ao
           conjunto de arestas admissíveis apenas as arestas que respeitam
           a restrição de salto H do problema
16 devolver T = (C, F)
```

forma a obter uma árvore de suporte aleatória que as respeite. Propõe-se o Algoritmo 7.1 que representa um algoritmo de cruzamento que efectua transformações necessárias para que as restrições de salto sejam satisfeitas, para o problema da *Árvore de Suporte de Custo Mínimo com Restrições de Salto* considerando a codificação dos cromossomas por sequências de arestas.

O Algoritmo 7.1, *PrimHMRST* (*Hop-constrained and Mutated Random Spanning Tree Prim based algorithm*), é baseado no Algoritmo 5.2, *PrimRST*. Contudo neste novo algoritmo a árvore resultante respeita as restrições de salto, uma vez que se efectua a operação da mutação em simultâneo com a operação de cruzamento. Os parâmetros de entrada do algoritmo consistem num grafo auxiliar (Secção 4.3), na identificação do vértice raiz, na indicação do valor da restrição de salto e de um conjunto com arestas para a operação de mutação. Este grafo auxiliar é composto pelas arestas dos cromossomas progenitores, adicionado de arestas do grafo do problema que não estão no grafo auxiliar. Estas últimas arestas constituem o conjunto B de arestas disponíveis para a operação de mutação.

O algoritmo inicia-se com a construção de um conjunto A com as arestas incidentes no vértice raiz (linha 4 do algoritmo). No entanto cada elemento deste conjunto é constituído por três números, um par de vértices que corresponde à aresta e um número que representa o salto associado a essa aresta. Inicialmente, as arestas incidentes no vértice raiz (nível 0), ligam a raiz a vértices no nível 1, logo o valor de salto associado a essas arestas é 1.

A operação da mutação é executada quando o conjunto B contém pelo menos uma aresta. Se o conjunto B está vazio então não serão executadas operações de mutação. Sempre que existe uma aresta no conjunto A que pertença ao conjunto B, essa aresta tem prioridade de selecção em relação a qualquer outra aresta (linha 6 e 7 do algoritmo), caso contrário é seleccionada, aleatoriamente, uma aresta do conjunto A (linha 9 e 10 do algoritmo). A aresta seleccionada é eliminada dos conjuntos A e B.

O passo seguinte consiste em verificar se essa aresta pode entrar para a árvore de suporte, de forma análoga ao método *PrimRST*. Se essa aresta for adicionada à árvore de suporte, é necessário actualizar o conjunto A com as arestas incidentes no vértice adicionado à árvore de suporte. As arestas incidentes no vértice adicionado à árvore de suporte, têm o salto incrementado em uma unidade em relação ao salto da aresta seleccionada, mas apenas são adicionadas ao conjunto A se respeitarem a restrição de salto do problema (linha 15 do algoritmo). Desta forma são geradas apenas árvores de suporte que respeitam a restrição de salto do problema.

No interesse de não sobrecarregar a notação do algoritmo devemos considerar o seguinte facto: embora os conjuntos A e B tenham elementos com dimensões diferentes, a intersecção destes conjuntos (linha 6) deve considerar apenas o par de números referentes às arestas e não considerar o número referente ao salto que faz parte da definição dos elementos do conjunto A.

Note-se que este método pode ser usado na geração de uma população que respeite a restrição de salto do problema. Para tal o grafo do problema deve ser usado no lugar do grafo auxiliar e o conjunto de arestas a mutar deve estar vazio (para que não ocorram mutações).

O Algoritmo 7.1, *PrimHMRST*, será exemplificado no Exemplo 7.1, e usa os conceitos de grafo conexo (Definição 2.8), árvore de suporte (Definição 2.10) e de salto (Definição 2.13).

Exemplo 7.1.

Aplicação do método de cruzamento e mutação PrimHMRST *ao grafo G_4, representado na Figura 7.1, seguindo os passos descritos no Algoritmo 5.2.*

Consideremos o problema representado pelo grafo K_6 da Figura 7.1, o valor de salto $H = 3$ e o vértice raiz representado pelo vértice 0. O grafo G_4 representa o grafo auxiliar resultante da união dos cromossomas pai e mãe, T_{11} e T_{12} respectivamente, também representados na Figura 7.1. A aresta representada a azul corresponde à aresta a introduzir pelo processo de mutação, $B = \{\{2, 1\}\}$. Note-se que esta aresta não pertence à reunião dos cromossomas progenitores, sendo uma aresta aleatória do problema.

Neste exemplo será aplicado o método de cruzamento e mutação *PrimHMRST* ao grafo auxiliar G_4, da Figura 7.1, de modo a obter uma das várias árvores de suporte aleatórias respeitando a restrição de salto do problema, que se podem obter por este método.

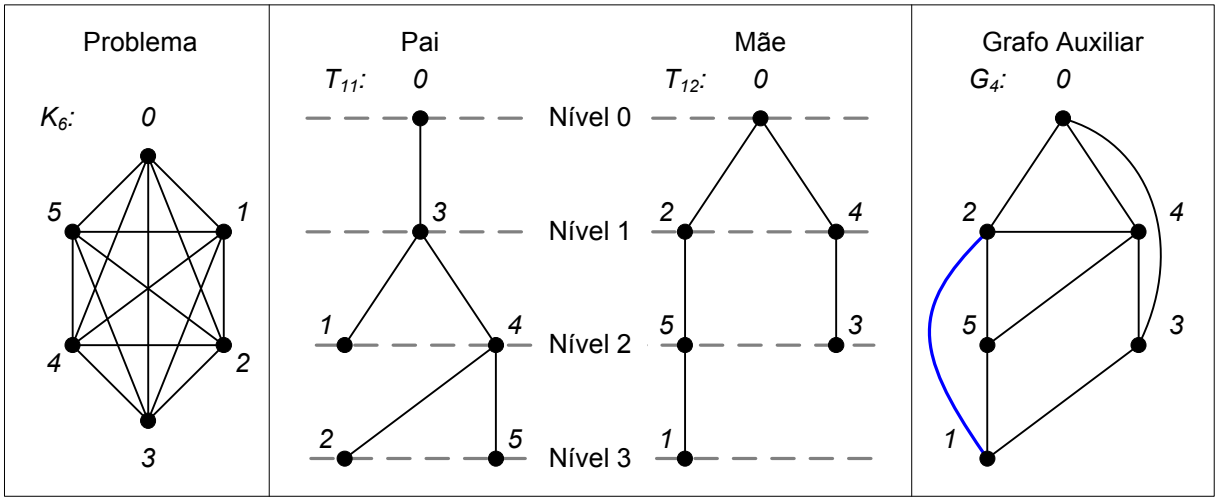

Figura 7.1: Representação do grafo de trabalho, cromossomas pai e mãe, e grafo auxiliar para o Exemplo 7.1. A aresta a azul está disponível para a operação de mutação.

A Tabela 7.1 apresenta os passos executados pelo Algoritmo 7.1, *PrimHMRST*, onde cada iteração representa o estado após a execução dos passos 6 a 15.

Início	$F \leftarrow \{\}$, $s \leftarrow 0$, $C \leftarrow \{0\}$, $A \leftarrow \{\{0,2,1\},\{0,3,1\},\{0,4,1\}\}$, $B \leftarrow \{\{2,1\}\}$
Iter. 1	$A \cap B = \{\}$, $B \leftarrow \{\{2,1\}\}$, $\{u,v,h\} \leftarrow \{0,4,1\}$, $4 \notin C$, $F \leftarrow F \cup \{0,4\}$, $C \leftarrow \{0,4\}$, $A \leftarrow \{\{0,2,1\},\{0,3,1\},\{4,2,2\},\{4,3,2\},\{4,5,2\}\}$
Iter. 2	$A \cap B = \{\}$, $B \leftarrow \{\{2,1\}\}$, $\{u,v,h\} \leftarrow \{0,3,1\}$, $3 \notin C$, $F \leftarrow F \cup \{0,3\}$, $C \leftarrow \{0,4,3\}$, $A \leftarrow \{\{0,2,1\},\{4,2,2\},\{4,3,2\},\{4,5,2\},\{3,1,2\}\}$
Iter. 3	$A \cap B = \{\}$, $B \leftarrow \{\{2,1\}\}$, $\{u,v,h\} \leftarrow \{4,3,2\}$, $3 \in C$, $C \leftarrow \{0,4,3\}$, $A \leftarrow \{\{0,2,1\},\{4,2,2\},\{4,5,2\},\{3,1,2\}\}$
Iter. 4	$A \cap B = \{\}$, $B \leftarrow \{\{2,1\}\}$, $\{u,v,h\} \leftarrow \{4,2,2\}$, $2 \notin C$, $F \leftarrow F \cup \{4,2\}$, $C \leftarrow \{0,4,3,2\}$, $A \leftarrow \{\{0,2,1\},\{4,5,2\},\{3,1,2\},\{2,1,3\},\{2,5,3\}\}$
Iter. 5	$A \cap B = \{2,1\}$, $B \leftarrow \{\}$, $\{u,v,h\} \leftarrow \{2,1,3\}$, $1 \notin C$, $F \leftarrow F \cup \{2,1\}$, $C \leftarrow \{0,4,3,2,1\}$, $A \leftarrow \{\{0,2,1\},\{4,5,2\},\{3,1,2\},\{2,5,3\}\}$ `//a aresta` $\{1,5\}$ `não entra em` A `porque o salto é superior a` $H = 3$
Iter. 6	$A \cap B = \{\}$, $B \leftarrow \{\}$, $\{u,v,h\} \leftarrow \{2,5,3\}$, $5 \notin C$, $F \leftarrow F \cup \{2,5\}$, $C \leftarrow \{0,4,3,2,1,5\}$, $A \leftarrow \{\{0,2,1\},\{4,5,2\},\{3,1,2\}\}$
Fim	$T = (C, F)$

Tabela 7.1: Representação dos passos descritos no Algoritmo 7.1 para o método de cruzamento e mutação *PrimHMRST*, considerando o grafo K_6 da Figura 7.1.

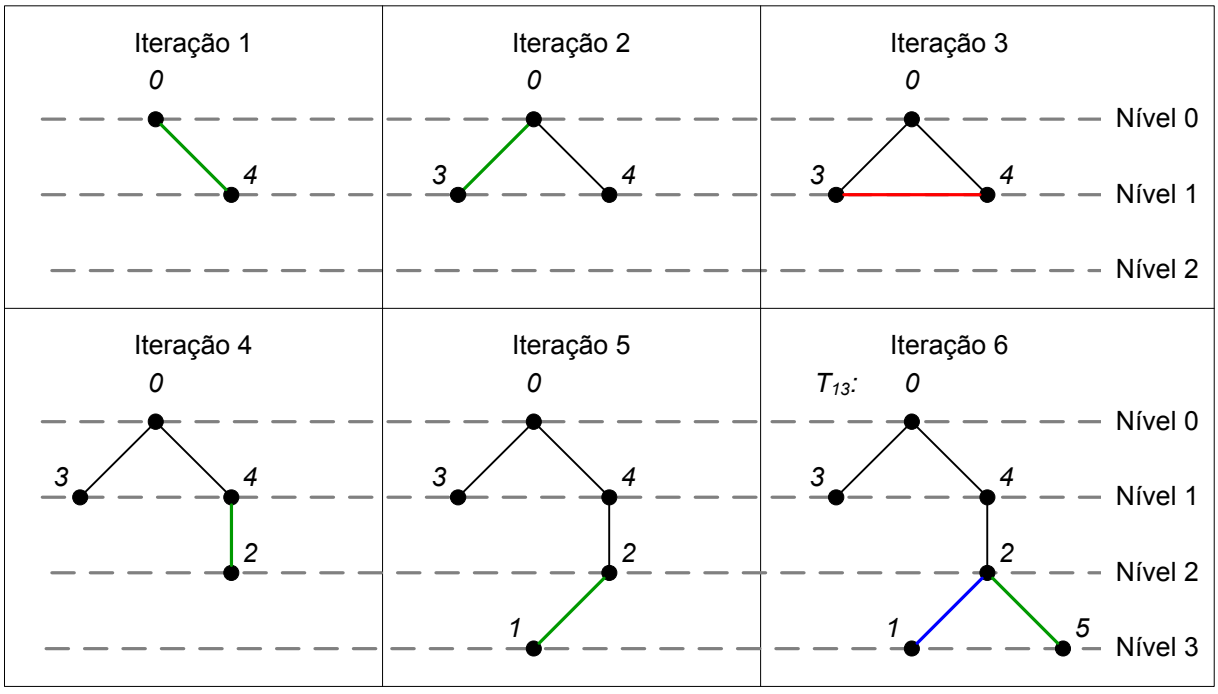

Figura 7.2: Representação gráfica dos passos executados na Tabela 7.1.

A tabela inclui uma iteração, a 3, onde a aresta seleccionada não pode ser adicionada à árvore de suporte porque formaria um ciclo. Na Figura 7.2 são representados graficamente os passos executados na Tabela 7.1. A verde está representada a aresta adicionada à árvore de suporte na respectiva iteração. A vermelho está representada a aresta que se adicionada à árvore de suporte da respectiva iteração formaria um ciclo, e, consequentemente, não é adicionada à árvore de suporte. A azul está representada a aresta correspondente ao processo de mutação.

A iteração 5, representada na Tabela 7.1, é de especial interesse, porque implementa o mecanismo da mutação e porque evita a entrada no conjunto A da aresta $\{1, 5\}$, cujo salto associado teria valor 4, valor superior ao valor da restrição de salto do exemplo.
A árvore de suporte gerada, T_{13}, é constituída pelo conjunto de arestas:

$$F = \{\{0, 4\}, \{0, 3\}, \{4, 2\}, \{2, 1\}, \{2, 5\}\}.$$

□

O método *PrimRST* foi escolhido para implementação (Capítulo 5), por forma a dar continuidade ao *Algoritmo de Prim*, introduzido no Capítulo 2. Da mesma forma o método *PrimRST* foi escolhido e modificado para respeitar as restrições de salto do problema da *Árvore de Suporte de Custo Mínimo com Restrições de Salto*, dando origem ao método *PrimHMRST* (Algoritmo 7.1).

O algoritmo *PrimHMRST* não foi implementado, sendo proposto como base para trabalhos futuros.

Capítulo 8

Considerações Finais

Neste trabalho apresentamos um algoritmo genético para o problema da *Árvore de Suporte de Custo Mínimo com Restrições de Salto*. Foram implementadas várias versões deste algoritmo genético, que diferem na codificação usada para os cromossomas, e nos métodos de geração, de cruzamento e de mutação dos indivíduos da população.

A codificação usada para os cromossomas consiste em duas codificações usadas noutros problemas com árvores de suporte com restrições, nomeadamente a codificação por sequências de Prüfer e a codificação por sequências de arestas. A utilização da codificação por sequências de Prüfer permite a utilização de métodos de cruzamento e mutação mais conhecidos, tais como o método *One Point Crossover* e o método *Single Point Mutation*, respectivamente. Por outro lado, a codificação por sequências de arestas necessita métodos específicos para as operações de cruzamento e de mutação, tendo sido implementado o método de cruzamento *PrimRST* e a primeira estratégia de mutação apresentada na Secção 4.3.

Estes métodos de cruzamento e de mutação, para ambas as codificações de cromossomas, geram indivíduos que podem não respeitar o valor da restrição de salto. Posteriormente, um teste de admissibilidade permite que apenas os indivíduos que respeitam o valor da restrição de salto sejam adicionados à população.

A população pode ser gerada por dois métodos. O primeiro método apresentado, recorre a uma geração aleatória da população, considerando apenas os indivíduos que respeitam o valor da restrição de salto. O segundo método apresentado corresponde à heurística *HRST*, que gera sempre indivíduos admissíveis, considerando o valor da restrição de salto.

Os testes computacionais revelaram que a geração aleatória da população apenas deve ser considerada para problemas com um número baixo de vértices e um valor de salto H

igual ou superior a 5. Os testes realizados para a instância TC1 com $N = 40$ e $H = 3,4$ revelaram que a geração aleatória da população é superior a 24 horas. Por outro lado, e nas mesmas condições, a geração da população pela heurística *HRST* realizou-se entre 24 a 27 segundos, para ambas as codificações em estudo.

Os testes computacionais revelaram, também, que a codificação por sequências de Prüfer nunca obteve resultados óptimos para nenhuma das instâncias. No entanto apresentou os melhores resultados para instâncias com um número de vértices superior a 21. A vantagem da codificação de Prüfer reside no método de cruzamento *One Point Crossover*. Este método tem um tempo de processamento muito baixo e gera dois indivíduos em cada iteração para cada par de progenitores. O método de cruzamento *PrimRST*, no mesmo tempo de processamento, gera apenas um indivíduo. Consequentemente o algoritmo genético com codificação por sequências de Prüfer consegue explorar no máximo, o "dobro" do espaço de pesquisa, em relação ao algoritmo genético com codificação por sequências de arestas. No entanto, o método de cruzamento *One Point Crossover* tem tendência a propagar o primeiro ou último elemento do cromossoma, impedindo o algoritmo genético com codificação de cromossomas por sequências de Prüfer de explorar todo o espaço de pesquisa, tornando-o mais dependente do processo de mutação. Verificamos que as percentagens de cruzamentos com sucesso são superiores para a codificação de Prüfer. Por fim, os tempos de processamento tendem a ser favoráveis para a codificação por sequências de arestas, porque não necessita codificar e descodificar cada cromossoma. A qualidade da solução tende a ser maior para a codificação por sequências de Prüfer para instâncias com mais de 21 vértices.

No final, apresentamos o método *PrimHMRST* como base para trabalhos futuros, para o problema da *Árvore de Suporte de Custo Mínimo com Restrições de Salto*. O algoritmo proposto considera simultaneamente, métodos de cruzamento e de mutação que respeitam a restrição de salto. Já tardiamente nos apercebemos da possibilidade que é apresentada neste método, não havendo tempo para a sua implementação e respectivos testes. Como trabalho futuro, podemos ainda sugerir a implementação das restrições de salto nos métodos *KruskalRST* e *RandWalkRST*. Ainda como trabalho futuro, podem ser efectuados melhoramentos à heurística *HRST*, para que considere a restrição $H \leq h$ em vez de considerar apenas $H = h$, onde $h \in \mathbb{N}$ representa o valor do salto. Em relação ao método de cruzamento *PrimRST*, podem ser retiradas as arestas do conjunto A (Algoritmo 5.2), na qual os vértices já se encontram na árvore de suporte (ver iterações 5 e 9 do Exemplo 5.2).

Glossário

CEMA Custo do elemento mais apto.

DP Dimensão da população.
DPCE Desvio padrão dos custos dos elementos.
DT Dimensão de um torneio.

Gap Valor do gap, corresponde à percentagem da diferença entre o custo obtido e o custo óptimo, $Gap = \frac{CEMA-OPT}{OPT} \times 100$ [17, 21].

H Valor do parâmetro da restrição de salto do problema.

MCE Média dos custos dos elementos.

N Ordem da instância.
NCT Número da configuração do teste.
NIRP Número de iterações para renovação da população.
NMI Número máximo de iterações do algoritmo genético.
NMM Número médio de filhos obtidos pela operação de mutação.
NT Número de torneios.
NTC Número total de filhos obtidos pela operação de cruzamento para cada execução do teste.

OPT Valor óptimo do problema.

PM Percentagem de mutação.

PMCI Percentagem de filhos obtidos pela operação de cruzamento sem sucesso.

PMCS Percentagem de filhos obtidos pela operação de cruzamento com sucesso.

PMMI Percentagem de filhos obtidos pela operação de mutação sem sucesso.

PMMS Percentagem de filhos obtidos pela operação de mutação com sucesso.

T Tempo de processamento, em segundos, utilizado para a execução do teste.

TM Tempo médio de processamento, em segundos, utilizado para cada execução do teste.

TT Tempo total de processamento, em minutos, utilizado para todas as execuções do teste.

Apêndice A

Resultados Computacionais Adicionais

Codificação por Sequências de Prüfer											
NCT	CEMA	MCE	DPCE	NTC	PMCS	PMCI	NMM	PMMS	PMMI	TM	TT
TE1, N=20, H=3, OPT=449											
1	484	567.4	38.1	20000	90.0	10.0	892.6	35.8	64.2	3.0	5.0
2	600	728.3	56.4	2000	91.6	8.4	90.8	43.1	56.9	0.0	0.0
3	658	716.0	24.5	20000	39.9	60.1	402.1	57.8	42.2	4.6	7.7
4	487	576.0	44.4	20000	96.3	3.7	965.6	30.6	69.4	3.0	5.0
5	487	581.8	54.6	20000	96.1	3.9	959.3	29.4	70.6	3.0	5.0
6	471	544.6	41.1	60000	94.7	5.3	2845.3	32.6	67.4	8.9	14.9
7	465	546.1	41.1	20000	88.9	11.1	3549.2	33.6	66.4	3.0	5.0
8	**463**	**521.2**	28.5	20000	69.8	30.2	13951.2	45.3	54.7	4.3	7.1
9	501	592.3	41.0	20000	60.4	39.6	602.9	50.5	49.5	3.0	5.0
10	583	690.0	33.5	20000	42.3	57.7	422.7	59.4	40.6	5.0	8.4
11	485	569.4	41.1	20000	85.7	14.3	856.2	34.2	65.8	3.0	5.0
12	471	552.7	43.9	20000	83.9	16.1	3360.3	33.0	67.0	3.0	5.1
TE1, N=20, H=4, OPT=385											
1	412	496.9	43.1	20000	94.3	5.8	942.5	51.0	49.0	3.0	5.0
2	541	673.0	55.3	2000	94.5	5.5	93.3	60.1	39.9	0.0	0.0
3	574	666.5	30.7	20000	56.5	43.5	563.3	72.4	27.6	4.9	8.2
4	420	512.0	49.4	20000	98.4	1.7	984.0	45.5	54.6	3.0	5.0
5	419	523.9	55.1	20000	99.0	1.0	990.7	45.7	54.3	3.0	5.0
6	403	485.7	37.9	60000	96.8	3.2	2910.5	46.3	53.8	9.0	15.0
7	414	484.5	41.5	20000	93.4	6.6	3725.1	48.6	51.4	3.1	5.1
8	**400**	**451.6**	28.8	20000	83.0	17.0	16593.6	55.5	44.5	4.4	7.3
9	429	511.7	40.5	20000	74.6	25.4	745.4	63.9	36.1	3.0	5.0
10	530	632.6	41.6	20000	58.8	41.2	585.7	72.8	27.2	5.0	8.3
11	418	502.9	47.1	20000	92.4	7.6	922.0	49.5	50.5	3.0	5.0
12	415	493.1	41.0	20000	91.2	8.8	3647.8	46.0	54.0	3.1	5.1
TE1, N=20, H=5, OPT=366											
1	387	465.5	40.0	20000	96.3	3.7	960.6	61.8	38.2	3.0	5.0
2	532	640.3	52.8	2000	96.6	3.4	94.5	72.1	27.9	0.0	0.0
3	543	635.8	31.3	20000	69.2	30.8	692.8	81.4	18.6	5.0	8.3
4	396	481.1	41.1	20000	99.1	1.0	989.4	60.1	39.9	3.0	5.0
5	401	492.5	48.1	20000	99.6	0.4	995.8	57.3	42.7	3.0	5.0
6	380	454.4	38.5	60000	98.0	2.0	2933.2	58.1	41.9	9.0	15.0
7	381	459.3	42.9	20000	95.8	4.2	3830.7	58.1	41.9	3.1	5.2
8	378	**420.3**	25.1	20000	89.9	10.1	17978.3	63.1	36.9	5.0	8.3
9	407	469.7	33.8	20000	84.0	16.1	838.7	72.5	27.5	3.0	5.0
10	493	611.4	44.9	20000	71.5	28.5	710.2	81.4	18.6	5.0	8.3
11	396	470.3	43.7	20000	95.7	4.3	953.9	61.8	38.2	3.0	5.0
12	**377**	457.6	40.5	20000	95.0	5.0	3786.8	57.6	42.4	3.2	5.3

Tabela 1: Representação dos resultados dos testes 1 a 12 para TE1 com $N = 20$ e codificação de cromossomas por sequências de Prüfer.

Codificação por Sequências de Arestas											
NCT	CEMA	MCE	DPCE	NTC	PMCS	PMCI	NMM	PMMS	PMMI	TM	TT
TE1, N=20, H=3, OPT=449											
1	562	644.8	40.3	10000	24.1	75.9	486.9	20.3	79.7	3.0	5.1
2	677	804.6	67.8	1000	25.0	75.0	48.8	21.7	78.3	0.0	0.0
3	670	750.7	25.8	10000	19.8	80.3	491.9	18.2	81.8	4.3	7.1
4	524	602.4	34.1	10000	66.4	33.6	449.1	42.2	57.8	2.9	4.9
5	552	629.2	38.8	10000	82.6	17.4	427.4	51.0	49.0	2.6	4.4
6	**476**	**523.3**	25.4	30000	39.5	60.5	1471.8	33.2	66.9	9.3	15.4
7	565	665.8	42.3	10000	21.5	78.5	1947.0	18.6	81.4	3.1	5.2
8	574	690.2	46.6	10000	16.5	83.5	10000.0	16.5	83.5	4.1	6.8
9	676	758.8	31.0	10000	17.9	82.1	496.3	16.5	83.5	4.0	6.7
10	671	737.2	21.5	10000	21.0	79.0	493.9	19.5	80.6	5.0	8.3
11	524	625.0	38.1	10000	18.8	81.3	484.8	14.5	85.6	3.9	6.4
12	545	622.6	33.8	10000	15.5	84.5	1940.1	12.4	87.6	4.0	6.7
TE1, N=20, H=4, OPT=385											
1	430	489.1	33.3	10000	41.0	59.0	487.5	37.1	63.0	3.0	5.0
2	533	657.2	53.6	1000	42.5	57.5	48.1	39.5	60.6	0.0	0.0
3	614	733.7	31.8	10000	37.4	62.6	492.7	35.8	64.2	4.9	8.1
4	417	508.6	31.0	10000	82.2	17.8	448.8	63.9	36.1	2.3	3.9
5	485	562.0	34.8	10000	93.5	6.5	409.4	73.2	26.8	2.1	3.5
6	**391**	**417.6**	12.0	30000	53.5	46.5	1472.2	47.9	52.1	9.0	15.0
7	433	493.4	29.4	10000	38.9	61.1	1921.3	36.2	63.8	3.0	5.0
8	443	492.9	20.3	10000	48.3	51.7	10000.0	48.3	51.7	4.0	6.7
9	630	706.8	28.8	10000	34.5	65.5	492.7	32.9	67.1	4.0	6.7
10	634	711.8	30.6	10000	40.2	59.8	496.0	38.6	61.4	5.0	8.3
11	402	475.0	28.0	10000	39.6	60.4	481.7	34.1	65.9	3.0	5.1
12	422	482.2	29.7	10000	37.0	63.0	1914.9	33.0	67.0	3.7	6.2
TE1, N=20, H=5, OPT=366											
1	380	409.8	15.0	10000	55.6	44.4	481.2	52.1	47.9	3.0	5.0
2	462	562.7	40.5	1000	54.5	45.6	49.0	52.9	47.1	0.0	0.0
3	629	707.4	27.8	10000	54.1	45.9	494.1	52.9	47.1	5.0	8.3
4	394	457.8	30.1	10000	89.5	10.5	458.5	75.7	24.3	2.2	3.6
5	425	519.1	39.6	10000	96.7	3.3	411.4	84.9	15.1	2.1	3.5
6	**367**	**380.0**	8.1	30000	66.0	34.0	1463.3	61.0	39.0	9.0	14.9
7	382	413.7	15.1	10000	55.6	44.4	1889.9	54.0	46.0	3.0	5.0
8	390	439.2	15.1	10000	71.8	28.2	10000.0	71.8	28.2	4.0	6.7
9	543	621.0	26.6	10000	50.4	49.6	494.4	49.1	50.9	4.0	6.7
10	573	676.4	34.4	10000	58.3	41.7	497.1	57.2	42.8	5.0	8.3
11	375	409.4	18.2	10000	58.6	41.4	479.6	53.0	47.0	3.0	5.0
12	384	412.8	14.6	10000	58.1	41.9	1905.4	55.1	44.9	3.1	5.1

Tabela 2: Representação dos resultados dos testes 1 a 12 para TE1 com $N = 20$ e codificação de cromossomas por sequências de arestas.

Codificação por Sequências de Prüfer											
NCT	CEMA	MCE	DPCE	NTC	PMCS	PMCI	NMM	PMMS	PMMI	TM	TT
TR1, N=20, H=3, OPT=168											
1	179	295.4	49.0	20000	96.4	3.6	959.7	50.2	49.8	3.0	5.0
2	294	481.5	71.3	2000	96.1	3.9	97.1	63.5	36.5	0.0	0.0
3	352	453.2	37.9	20000	68.9	31.1	687.2	73.9	26.1	5.0	8.4
4	209	317.5	60.9	20000	98.8	1.2	988.6	46.4	53.6	3.0	5.0
5	216	326.4	63.9	20000	99.2	0.8	986.4	44.8	55.2	3.0	5.1
6	**174**	275.5	47.5	60000	98.0	2.0	2941.2	46.4	53.7	9.0	15.0
7	187	275.7	41.5	20000	96.0	4.0	3833.0	49.1	50.9	3.3	5.5
8	180	**235.6**	25.9	20000	89.2	10.8	17833.9	53.4	46.6	5.2	8.7
9	192	279.8	35.4	20000	83.6	16.4	833.7	62.5	37.5	3.1	5.2
10	317	423.4	46.0	20000	66.7	33.3	663.6	72.3	27.7	5.2	8.7
11	201	304.5	57.9	20000	94.5	5.5	945.6	48.6	51.4	3.1	5.2
12	177	285.2	51.3	20000	93.7	6.3	3750.2	45.5	54.5	3.9	6.5
TR1, N=20, H=4, OPT=146											
1	183	279.4	45.9	20000	97.7	2.3	974.2	67.5	32.5	3.0	5.0
2	334	470.7	63.6	2000	97.4	2.6	98.5	75.6	24.4	0.0	0.0
3	349	426.9	36.7	20000	78.5	21.5	782.9	83.8	16.3	5.0	8.4
4	177	296.8	53.5	20000	99.3	0.7	992.4	62.8	37.2	3.0	5.0
5	190	304.1	55.2	20000	99.7	0.3	997.6	60.4	39.6	3.0	5.1
6	159	263.0	42.6	60000	98.8	1.3	2954.4	64.4	35.6	9.0	15.0
7	**156**	264.3	41.6	20000	97.5	2.5	3887.5	65.0	35.0	3.4	5.7
8	**156**	**222.7**	29.0	20000	94.0	6.0	18805.0	68.6	31.4	5.6	9.4
9	206	261.0	33.4	20000	89.8	10.2	896.4	76.9	23.1	3.2	5.3
10	301	400.9	55.0	20000	77.1	22.9	767.7	82.6	17.4	5.1	8.5
11	169	270.8	50.6	20000	96.9	3.1	965.9	66.5	33.5	3.1	5.2
12	166	277.2	46.4	20000	96.6	3.4	3850.5	64.2	35.8	4.0	6.7
TR1, N=20, H=5, OPT=137											
1	160	270.2	46.1	20000	98.5	1.5	986.7	78.9	21.1	3.0	5.0
2	263	451.3	65.1	2000	98.3	1.7	99.2	85.8	14.2	0.0	0.0
3	293	416.2	41.8	20000	84.3	15.7	844.3	88.9	11.1	5.1	8.4
4	187	294.0	50.7	20000	99.6	0.5	989.4	74.6	25.4	3.0	5.0
5	202	297.5	54.0	20000	99.8	0.2	998.4	73.4	26.6	3.1	5.1
6	172	255.9	41.4	60000	99.2	0.8	2969.1	77.3	22.7	9.0	15.0
7	156	253.6	42.3	20000	98.3	1.8	3936.5	75.2	24.8	3.4	5.7
8	**154**	**220.9**	29.8	20000	96.5	3.5	19293.5	79.6	20.4	5.1	8.5
9	165	259.7	42.6	20000	93.0	7.0	933.8	84.5	15.5	3.1	5.2
10	263	383.7	53.6	20000	83.9	16.1	836.2	88.3	11.7	5.2	8.7
11	175	266.9	44.7	20000	98.0	2.0	970.8	77.0	23.0	3.1	5.2
12	181	264.6	41.1	20000	97.9	2.1	3917.0	75.8	24.2	4.0	6.7

Tabela 3: Representação dos resultados dos testes 1 a 12 para TR1 com $N = 20$ e codificação de cromossomas por sequências de Prüfer.

Codificação por Sequências de Arestas											
NCT	CEMA	MCE	DPCE	NTC	PMCS	PMCI	NMM	PMMS	PMMI	TM	TT
TR1, N=20, H=3, OPT=168											
1	177	235.9	25.2	10000	64.3	35.7	497.1	55.1	44.9	3.0	5.0
2	302	420.2	53.2	1000	64.3	35.8	48.9	58.3	41.7	0.0	0.0
3	516	593.9	31.5	10000	46.1	53.9	497.8	43.6	56.4	4.1	6.9
4	235	315.3	40.3	10000	88.3	11.7	482.2	61.5	38.5	2.2	3.6
5	281	379.8	50.7	10000	93.6	6.5	476.2	64.5	35.6	2.1	3.6
6	**168**	**197.6**	16.3	30000	73.8	26.2	1485.1	61.8	38.2	8.0	13.4
7	173	243.8	24.9	10000	58.6	41.4	1979.6	52.2	47.8	3.0	5.0
8	264	342.0	33.3	10000	41.5	58.5	10000.0	41.5	58.5	4.0	6.7
9	380	499.9	38.8	10000	49.8	50.2	494.5	46.9	53.1	3.2	5.3
10	442	560.3	42.3	10000	42.6	57.4	491.6	40.2	59.8	5.0	8.3
11	188	243.1	23.6	10000	59.5	40.5	494.0	47.8	52.2	3.0	5.0
12	192	254.8	32.6	10000	52.0	48.0	1965.7	43.5	56.5	3.0	5.1
TR1, N=20, H=4, OPT=146											
1	150	179.2	17.3	10000	74.6	25.4	493.1	68.5	31.5	3.0	5.0
2	225	347.4	45.9	1000	73.9	26.1	50.3	70.3	29.7	0.0	0.0
3	389	538.5	36.1	10000	65.8	34.2	499.0	63.9	36.1	4.9	8.1
4	177	262.5	37.8	10000	92.8	7.2	487.1	73.4	26.6	2.2	3.6
5	236	337.1	45.1	10000	96.2	3.8	475.6	76.5	23.5	2.1	3.6
6	**146**	**157.2**	8.3	30000	80.7	19.4	1473.1	71.9	28.1	8.0	13.4
7	151	182.0	17.7	10000	70.4	29.6	1974.0	65.8	34.2	3.0	5.0
8	199	252.5	26.1	10000	59.2	40.8	10000.0	59.2	40.8	4.0	6.7
9	290	409.5	33.7	10000	67.7	32.3	493.4	65.7	34.3	3.3	5.5
10	407	512.3	44.5	10000	63.5	36.6	496.6	61.7	38.3	5.0	8.3
11	150	181.7	17.5	10000	75.0	25.1	494.2	65.3	34.7	3.0	5.0
12	150	189.9	19.3	10000	69.0	31.0	1964.6	61.9	38.1	3.0	5.0
TR1, N=20, H=5, OPT=137											
1	**137**	151.0	11.5	10000	81.4	18.6	495.9	75.4	24.7	3.0	5.0
2	207	314.9	44.8	1000	81.6	18.4	48.5	79.7	20.3	0.0	0.0
3	420	509.2	32.9	10000	78.8	21.2	496.9	77.5	22.5	5.0	8.3
4	157	235.0	35.5	10000	95.2	4.8	488.9	81.9	18.1	2.1	3.5
5	222	311.7	42.1	10000	97.6	2.4	476.3	85.6	14.4	2.1	3.6
6	**137**	**141.3**	7.6	30000	86.5	13.5	1492.3	77.8	22.2	8.0	13.3
7	**137**	157.0	12.7	10000	78.7	21.3	1979.5	75.4	24.6	3.0	5.0
8	159	209.6	16.4	10000	72.4	27.6	10000.0	72.4	27.6	4.0	6.7
9	295	354.1	30.5	10000	78.8	21.2	498.4	77.4	22.6	3.4	5.7
10	374	482.0	36.9	10000	77.7	22.4	499.8	76.4	23.6	5.0	8.3
11	**137**	155.8	13.7	10000	83.1	16.9	495.5	74.8	25.2	3.0	5.0
12	**137**	158.4	13.4	10000	79.2	20.8	1973.5	73.8	26.2	3.0	5.0

Tabela 4: Representação dos resultados dos testes 1 a 12 para TR1 com $N = 20$ e codificação de cromossomas por sequências de arestas.

Codificação por Sequências de Prüfer												
N	H	OPT	CEMA	Gap	MCE	DPCE	PMCS	PMCI	PMMS	PMMI	TM	TT
20	3	449	471	4.9	544.6	41.1	94.7	5.3	32.6	67.4	8.9	14.9
	4	385	403	4.7	485.7	37.9	96.8	3.2	46.3	53.8	9.0	15.0
	5	366	380	3.8	454.4	38.5	98.0	2.0	58.1	41.9	9.0	15.0
40	3	708	826	16.7	1065.2	79.8	92.8	7.2	35.9	64.1	26.6	44.4
	4	627	758	20.9	962.4	74.7	95.6	4.4	47.6	52.4	27.1	45.1
	5	590	776	31.5	895.9	66.0	97.3	2.7	58.4	41.6	26.9	44.8
60	3	1525	2504	64.2	2880.6	203.5	90.4	9.6	34.7	65.3	55.0	91.7
	4	1336	2179	63.1	2582.7	204.2	93.8	6.2	46.2	53.8	56.2	93.6
	5	1225	1951	59.3	2387.3	210.3	96.2	3.8	55.5	44.5	57.1	95.1
80	3	1806	3504	94.0	4151.6	306.1	89.6	10.4	35.0	65.0	96.5	160.9
	4	1558	3024	94.1	3673.1	312.5	93.0	7.0	47.6	52.4	100.8	168.0
	5	1442	2849	97.6	3391.5	278.5	95.6	4.5	55.6	44.4	95.6	159.3

Tabela 5: Teste 6 para as instâncias TE1 com $N = 20, 40, 60, 80$ e codificação de cromossomas por sequências de Prüfer.

Codificação por Sequências de Arestas												
N	H	OPT	CEMA	Gap	MCE	DPCE	PMCS	PMCI	PMMS	PMMI	TM	TT
20	3	449	476	6.0	523.3	25.4	39.5	60.5	33.2	66.9	9.3	15.4
	4	385	391	1.6	417.6	12.0	53.5	46.5	47.9	52.1	9.0	15.0
	5	366	367	0.3	380.0	8.1	66.0	34.0	61.0	39.0	9.0	14.9
40	3	708	1096	54.8	1315.0	107.9	28.2	71.8	25.6	74.4	29.0	48.3
	4	627	862	37.5	970.8	54.0	44.2	55.8	41.6	58.4	27.9	46.4
	5	590	733	24.2	822.4	44.9	53.7	46.3	51.2	48.8	27.0	45.0
60	3	1525	3580	134.8	4279.3	296.1	16.5	83.5	15.1	84.9	58.0	96.7
	4	1336	2661	99.2	3170.4	288.3	38.3	61.7	36.7	63.3	54.3	90.5
	5	1225	2239	82.8	2661.7	230.0	52.2	47.8	50.5	49.5	52.2	86.9
80	3	1806	5214	188.7	6247.4	340.3	12.2	87.8	11.4	88.6	99.0	165.1
	4	1558	4011	157.4	4870.3	449.5	37.8	62.2	36.1	63.9	90.9	151.5
	5	1442	3291	128.2	4132.9	417.6	54.2	45.8	52.9	47.2	88.5	147.5

Tabela 6: Teste 6 para as instâncias TE1 com $N = 20, 40, 60, 80$ e codificação de cromossomas por sequências de arestas.

Codificação por Sequências de Prüfer												
N	H	OPT	CEMA	Gap	MCE	DPCE	PMCS	PMCI	PMMS	PMMI	TM	TT
20	3	168	174	3.6	275.5	47.5	98.0	2.0	46.4	53.7	9.0	15.0
	4	146	159	8.9	263.0	42.6	98.8	1.3	64.4	35.6	9.0	15.0
	5	137	172	25.5	255.9	41.4	99.2	0.8	77.3	22.7	9.0	15.0
40	3	176	451	156.3	598.6	68.5	96.9	3.1	38.2	61.8	26.6	44.3
	4	149	370	148.3	519.7	74.4	98.0	2.0	51.5	48.5	26.8	44.7
	5	139	364	161.9	475.4	61.1	98.7	1.3	62.5	37.5	27.1	45.2
60	3	213	844	296.2	1121.1	143.9	93.4	6.6	35.6	64.4	56.1	93.5
	4	152	730	380.3	956.6	120.9	95.6	4.4	47.5	52.5	56.3	93.8
	5	124	601	384.7	861.0	108.0	97.3	2.8	58.2	41.9	57.0	94.9
80	3	208	857	312.0	1104.3	125.6	97.7	2.3	44.1	55.9	88.1	146.8
	4	180	678	276.7	990.7	129.6	98.5	1.5	55.9	44.1	92.1	153.6
	5	164	694	323.2	937.2	115.2	99.0	1.0	65.3	34.7	103.4	172.3

Tabela 7: Teste 6 para as instâncias TR1 com $N = 20, 40, 60, 80$ e codificação de cromossomas por sequências de Prüfer.

Codificação por Sequências de Arestas												
N	H	OPT	CEMA	Gap	MCE	DPCE	PMCS	PMCI	PMMS	PMMI	TM	TT
20	3	168	168	0.0	197.6	16.3	73.8	26.2	61.8	38.2	8.0	13.4
	4	146	146	0.0	157.2	8.3	80.7	19.4	71.9	28.1	8.0	13.4
	5	137	137	0.0	141.3	7.6	86.5	13.5	77.8	22.2	8.0	13.3
40	3	176	410	133.0	538.0	51.1	65.0	35.1	58.2	41.8	25.3	42.1
	4	149	298	100.0	367.4	35.9	70.2	29.8	65.9	34.1	25.4	42.4
	5	139	229	64.7	280.4	26.2	76.4	23.6	73.1	26.9	25.5	42.4
60	3	213	1005	371.8	1395.2	91.1	55.0	45.1	51.5	48.5	50.2	83.7
	4	152	866	469.7	1046.6	79.2	65.1	34.9	62.7	37.3	50.3	83.8
	5	124	640	416.1	831.5	74.0	73.4	26.6	71.7	28.3	49.7	82.9
80	3	208	1238	495.2	1420.7	81.5	76.8	23.2	72.1	27.9	72.6	120.9
	4	180	939	421.7	1156.2	82.8	85.0	15.0	83.1	16.9	74.0	123.3
	5	164	861	425.0	1000.2	70.2	91.3	8.7	90.4	9.6	74.2	123.7

Tabela 8: Teste 6 para as instâncias TR1 com $N = 20, 40, 60, 80$ e codificação de cromossomas por sequências de arestas.

Apêndice B

Documentação da Implementação do Algoritmo Genético em Java para a Codificação por Sequências de Prüfer

Capítulo 1

Package pc.dissertacao.ua.runtime

Package Contents *Page*

Classes

1.1 Classes

1.1.1 Class **Consola**

Class main do projecto.

Declaration

public class Consola **extends** java.lang.Object

Constructors

- *Consola*
  `public` **Consola**`( )`

Methods

- *main*
  `public static void` **main**`( java.lang.String [] args )`
  - **Parameters**
    - `args` - Argumentos passados pela linha de comandos

1.1.2 Class **Ficheiro**

Esta classe implementa mecanismos de leitura de ficheiros com matrizes de adjacência para a construção/inicialização dos problemas, e permite guardar os resultados dos testes efectuados.

Declaration

public class Ficheiro **extends** java.lang.Object

Fields

- public static final int PARAMETRO_CUSTO
  - Constante que indica que o parâmetro de trabalho é um parâmetro de custo

Constructors

- *Ficheiro*
  `public` **Ficheiro**`( )`

METHODS

- *escreverResultadosCSV*
  `public static boolean` **escreverResultadosCSV**`( java.lang.String` **nomeDoFicheiro**`, java.lang.String` **resultados** `)`
  - **Usage**
    - Permite guardar os dados de uma sessão de testes num ficheiro separado por ponto-e-virgulas ';'. Os dados guardados são: Número de cada teste, Tempo de cada teste, Número de cruzamentos com sucesso de cada teste, Número de cruzamentos sem sucesso de cada teste, Número de mutações com sucesso de cada teste, Número de mutações sem sucesso de cada teste, O elemento mais apto de cada teste, O custo do elemento mais apto de cada teste.
  - **Parameters**
    - `nomeDoFicheiro` - Nome do ficheiro a guardar os dados do teste
    - `resultados` - Resultados do teste
  - **Returns** - Indica se os dados foram guardados com sucesso no ficheiro

---

- *escreverResultadosExcel*
  `public static boolean` **escreverResultadosExcel**`( java.lang.String` **nomeDoFicheiro**`, java.lang.String` **nomeFolhaExcel**`, int` **numLinha**`, int` **tipoASCM**`, boolean` **utilizaHeuristica**`, boolean` **ultimoTeste**`, int` **objectivo**`, int` **IDTeste** `)`
  - **Usage**
    - Permite guardar os dados de uma sessão de testes num ficheiro com formato excel.
  - **Parameters**
    - `nomeDoFicheiro` - Nome do ficheiro a guardar os dados do teste
    - `nomeFolhaExcel` - Nome da folha de excel a guardar
    - `numLinha` - Número de execução do teste actual
    - `tipoASCM` - Tipo de árvore de suporte de custo mínimo
    - `utilizaHeuristica` - Indica se é utilizada a heurística para geração da população
    - `ultimoTeste` - Indica se é o último teste
    - `objectivo` - Indica qual o objectivo do teste (minimização ou maximização)
    - `IDTeste` - Indica qual o teste que está a ser guardado
  - **Returns** - Indica se os dados foram guardados com sucesso no ficheiro excel

---

- *escreverResultadosIter*
  `public static boolean` **escreverResultadosIter**`( )`
  - **Usage**
    - Permite guardar os dados de cada iteração num num ficheiro separado por ponto-e-virgulas ';'. Os dados guardados são: Número de cada iteração, O custo do elemento mais apto de cada iteração, Tempo de cada iteração.
  - **Returns** - Indica se os dados foram guardados com sucesso no ficheiro

---

- *escreverResumoExcel*
  `public static boolean` **escreverResumoExcel**`( java.lang.String` **nomeDoFicheiro**`, int` **numRepeticoes** `)`
  - **Usage**
    - Permite gerar uma folha excel com o resumo dos testes efectuados

  - **Parameters**
    - `nomeDoFicheiro` - Nome do ficheiro a guardar os dados do teste
    - `numRepeticoes` - Número total de execução de cada teste
  - **Returns** - Indica se o resumo foi guardado com sucesso no ficheiro excel

---

- *lerFicheiroMatrizDeAdjacenciaTriangularSuperior*
  `public static boolean` **lerFicheiroMatrizDeAdjacenciaTriangularSuperior**`( int` **tipoParametro**`, java.lang.String` **enderecoDoFicheiro**`, pc.dissertacao.ua.grafos.Grafo` **grafo**`, boolean` **debug**`, int` **numLinhasPorLinhaDaMatriz**`, int` **limiteDeVertices** `)`

  - **Usage**
    - Permite ler um ficheiro com uma matriz de adjacência no qual os dados relevantes estão na parte triangular superior da matriz
  - **Parameters**
    - `tipoParametro` - Indica o tipo de parâmetro a ler (ver constantes acima)
    - `enderecoDoFicheiro` - Endereço do ficheiro a ler
    - `grafo` - Grafo para adicionar os vértices e arestas lidas
    - `debug` - Escreve na consola todos os outputs (caso true)
    - `numLinhasPorLinhaDaMatriz` - Indica quantas linhas no ficheiro correspondem a uma linha na matriz de adjacência
    - `limiteDeVertices` - Indica se o ficheiro é lido na totalidade (0) ou se é forçada a obtenção de um grafo não completo com um determinado número de vértices
  - **Returns** - Indica se o ficheiro foi lido com sucesso

1.1.3 Class **Sondagem**

Esta classe permite registar os dados de um teste (uma execução completa do algoritmo evolutivo) para posterior armazenamento num ficheiro.

Declaration

public class Sondagem **extends** java.lang.Object

Fields

- public static int NUMERO_DO_TESTE
  - Constante que indica o número do teste
- public static long TEMPO_DO_TESTE
  - Constante que indica o tempo do teste
- public static long NUM_CRUZAMENTOS_COM_SUCESSO
  - Constante que indica o número de cruzamentos com sucesso no teste
- public static long NUM_CRUZAMENTOS_SEM_SUCESSO

  - Constante que indica o número de cruzamentos sem sucesso no teste
- public static long NUM_MUTACOES_COM_SUCESSO
  - Constante que indica o número de mutações com sucesso no teste
- public static long NUM_MUTACOES_SEM_SUCESSO
  - Constante que indica o número de mutações sem sucesso no teste
- public static String ELEMENTO_MAIS_APTO
  - Constante que indica o elemento mais apto do teste
- public static int CUSTO_ELEMENTO_MAIS_APTO
  - Constante que indica o custo do elemento mais apto do teste
- public static Vector ELEMENTO_MAIS_APTO_ITER
  - Variável que guarda o custo do elemento mais apto do teste por iteração
- public static Vector TEMPO_ITER
  - Variável que guarda o tempo de cada iteração

CONSTRUCTORS

- *Sondagem*
  `public` **Sondagem**`( )`

METHODS

- *imprimirSondagem*
  `public static void` **imprimirSondagem**`( )`
  - **Usage**
    - Permite imprimir os dados recolhidos durante a execução do algoritmo evolutivo

---

- *obterSondagem*
  `public static String` **obterSondagem**`( )`
  - **Usage**
    - Permite obter os resultados de uma execução do algoritmo evolutivo no formato usado para armazenamento no ficheiro.
  - **Returns** - resultados de uma execução do algoritmo evolutivo

---

- *repor*
  `public static void` **repor**`( )`
  - **Usage**
    - Repõe os valores da sondagem a zero. Deve ser chamado após cada execução completa do algoritmo evolutivo.

1.1.4 Class **Testes**

Classe que permite criar conjuntos de teste para execução em batch (bloco)

Declaration

public class Testes
extends java.lang.Object

Constructors

- *Testes*
  `public` **Testes**`( )`

Methods

- *resumoExcel*
  `public static void` **resumoExcel**`( java.lang.String` **nomeDoFicheiro**`, int` **numRepeticoes** `)`
  - **Usage**
    - Permite gerar uma folha excel com o resumo dos testes efectuados
  - **Parameters**
    - `nomeDoFicheiro` - Nome do ficheiro a guardar os dados do teste
    - `numRepeticoes` - Número total de execução de cada teste

- *Teste1*
  `public static void` **Teste1**`( java.lang.String` **nomeDoFicheiro**`, int` **numLinhasPorLinhaDaMatriz**`, boolean` **usarHeuristicaGeracaoPopulacao**`, int` **numTestes** `)`
  - **Usage**
    - Permite executar o conjunto de testes associados ao Teste 1
  - **Parameters**
    - `nomeDoFicheiro` - Nome do ficheiro de dados
    - `numLinhasPorLinhaDaMatriz` - Indica quantas linhas no ficheiro correspondem a uma linha na matriz de adjacência
    - `usarHeuristicaGeracaoPopulacao` - Usar heurística para geração da população
    - `numTestes` - Número de testes a efectuar

- *Teste10*
  `public static void` **Teste10**`( java.lang.String` **nomeDoFicheiro**`, int` **numLinhasPorLinhaDaMatriz**`, boolean` **usarHeuristicaGeracaoPopulacao**`, int` **numTestes** `)`
  - **Usage**
    - Permite executar o conjunto de testes associados ao Teste 10
  - **Parameters**

    * `nomeDoFicheiro` - Nome do ficheiro de dados
    * `numLinhasPorLinhaDaMatriz` - Indica quantas linhas no ficheiro correspondem a uma linha na matriz de adjacência
    * `usarHeuristicaGeracaoPopulacao` - Usar heurística para geração da população
    * `numTestes` - Número de testes a efectuar

- *Teste11*
  `public static void` **Teste11**`( java.lang.String` **nomeDoFicheiro**`, int` **numLinhasPorLinhaDaMatriz**`, boolean` **usarHeuristicaGeracaoPopulacao**`, int` **numTestes** `)`

  – **Usage**
    * Permite executar o conjunto de testes associados ao Teste 11

  – **Parameters**
    * `nomeDoFicheiro` - Nome do ficheiro de dados
    * `numLinhasPorLinhaDaMatriz` - Indica quantas linhas no ficheiro correspondem a uma linha na matriz de adjacência
    * `usarHeuristicaGeracaoPopulacao` - Usar heurística para geração da população
    * `numTestes` - Número de testes a efectuar

- *Teste12*
  `public static void` **Teste12**`( java.lang.String` **nomeDoFicheiro**`, int` **numLinhasPorLinhaDaMatriz**`, boolean` **usarHeuristicaGeracaoPopulacao**`, int` **numTestes** `)`

  – **Usage**
    * Permite executar o conjunto de testes associados ao Teste 12

  – **Parameters**
    * `nomeDoFicheiro` - Nome do ficheiro de dados
    * `numLinhasPorLinhaDaMatriz` - Indica quantas linhas no ficheiro correspondem a uma linha na matriz de adjacência
    * `usarHeuristicaGeracaoPopulacao` - Usar heurística para geração da população
    * `numTestes` - Número de testes a efectuar

- *Teste2*
  `public static void` **Teste2**`( java.lang.String` **nomeDoFicheiro**`, int` **numLinhasPorLinhaDaMatriz**`, boolean` **usarHeuristicaGeracaoPopulacao**`, int` **numTestes** `)`

  – **Usage**
    * Permite executar o conjunto de testes associados ao Teste 2

  – **Parameters**
    * `nomeDoFicheiro` - Nome do ficheiro de dados
    * `numLinhasPorLinhaDaMatriz` - Indica quantas linhas no ficheiro correspondem a uma linha na matriz de adjacência
    * `usarHeuristicaGeracaoPopulacao` - Usar heurística para geração da população
    * `numTestes` - Número de testes a efectuar

- *Teste3*
  `public static void` **Teste3**`( java.lang.String` **nomeDoFicheiro**`, int` **numLinhasPorLinhaDaMatriz**`, boolean` **usarHeuristicaGeracaoPopulacao**`, int` **numTestes** `)`

  – **Usage**

    * Permite executar o conjunto de testes associados ao Teste 3
  - **Parameters**
    * `nomeDoFicheiro` - Nome do ficheiro de dados
    * `numLinhasPorLinhaDaMatriz` - Indica quantas linhas no ficheiro correspondem a uma linha na matriz de adjacência
    * `usarHeuristicaGeracaoPopulacao` - Usar heurística para geração da população
    * `numTestes` - Número de testes a efectuar

- *Teste4*
  `public static void` **Teste4**`( java.lang.String` **nomeDoFicheiro**`, int` **numLinhasPorLinhaDaMatriz**`, boolean` **usarHeuristicaGeracaoPopulacao**`, int` **numTestes** `)`
  - **Usage**
    * Permite executar o conjunto de testes associados ao Teste 4
  - **Parameters**
    * `nomeDoFicheiro` - Nome do ficheiro de dados
    * `numLinhasPorLinhaDaMatriz` - Indica quantas linhas no ficheiro correspondem a uma linha na matriz de adjacência
    * `usarHeuristicaGeracaoPopulacao` - Usar heurística para geração da população
    * `numTestes` - Número de testes a efectuar

- *Teste5*
  `public static void` **Teste5**`( java.lang.String` **nomeDoFicheiro**`, int` **numLinhasPorLinhaDaMatriz**`, boolean` **usarHeuristicaGeracaoPopulacao**`, int` **numTestes** `)`
  - **Usage**
    * Permite executar o conjunto de testes associados ao Teste 5
  - **Parameters**
    * `nomeDoFicheiro` - Nome do ficheiro de dados
    * `numLinhasPorLinhaDaMatriz` - Indica quantas linhas no ficheiro correspondem a uma linha na matriz de adjacência
    * `usarHeuristicaGeracaoPopulacao` - Usar heurística para geração da população
    * `numTestes` - Número de testes a efectuar

- *Teste6*
  `public static void` **Teste6**`( java.lang.String` **nomeDoFicheiro**`, int` **numLinhasPorLinhaDaMatriz**`, boolean` **usarHeuristicaGeracaoPopulacao**`, int` **numTestes** `)`
  - **Usage**
    * Permite executar o conjunto de testes associados ao Teste 6
  - **Parameters**
    * `nomeDoFicheiro` - Nome do ficheiro de dados
    * `numLinhasPorLinhaDaMatriz` - Indica quantas linhas no ficheiro correspondem a uma linha na matriz de adjacência
    * `usarHeuristicaGeracaoPopulacao` - Usar heurística para geração da população
    * `numTestes` - Número de testes a efectuar

- *Teste7*
  `public static void` **Teste7**`( java.lang.String` **nomeDoFicheiro**`, int` **numLinhasPorLinhaDaMatriz**`, boolean` **usarHeuristicaGeracaoPopulacao**`, int` **numTestes** `)`

  - **Usage**
    * Permite executar o conjunto de testes associados ao Teste 7
  - **Parameters**
    * `nomeDoFicheiro` - Nome do ficheiro de dados
    * `numLinhasPorLinhaDaMatriz` - Indica quantas linhas no ficheiro correspondem a uma linha na matriz de adjacência
    * `usarHeuristicaGeracaoPopulacao` - Usar heurística para geração da população
    * `numTestes` - Número de testes a efectuar

- *Teste8*
`public static void` **Teste8**`( java.lang.String` **nomeDoFicheiro**`, int` **numLinhasPorLinhaDaMatriz**`, boolean` **usarHeuristicaGeracaoPopulacao**`, int` **numTestes** `)`

  - **Usage**
    * Permite executar o conjunto de testes associados ao Teste 8
  - **Parameters**
    * `nomeDoFicheiro` - Nome do ficheiro de dados
    * `numLinhasPorLinhaDaMatriz` - Indica quantas linhas no ficheiro correspondem a uma linha na matriz de adjacência
    * `usarHeuristicaGeracaoPopulacao` - Usar heurística para geração da população
    * `numTestes` - Número de testes a efectuar

- *Teste9*
`public static void` **Teste9**`( java.lang.String` **nomeDoFicheiro**`, int` **numLinhasPorLinhaDaMatriz**`, boolean` **usarHeuristicaGeracaoPopulacao**`, int` **numTestes** `)`

  - **Usage**
    * Permite executar o conjunto de testes associados ao Teste 9
  - **Parameters**
    * `nomeDoFicheiro` - Nome do ficheiro de dados
    * `numLinhasPorLinhaDaMatriz` - Indica quantas linhas no ficheiro correspondem a uma linha na matriz de adjacência
    * `usarHeuristicaGeracaoPopulacao` - Usar heurística para geração da população
    * `numTestes` - Número de testes a efectuar

Capítulo 2

Package pc.dissertacao.ua.algoevo

Package Contents *Page*

Classes

2.1 Classes

2.1.1 Class **Alelo**

Esta classe implementa mecanismos que definem um alelo de um cromossoma

Declaration

public class Alelo **extends** java.lang.Object

Constructors

- *Alelo*
  `public` **Alelo**`( int` **alelo** `)`
  - **Usage**
    - Cria uma instância da classe Alelo
  - **Parameters**
    - `alelo` - Valor do alelo

Methods

- *definirAlelo*
  `public void` **definirAlelo**`( int` **alelo** `)`
  - **Usage**
    - Permite definir o valor do alelo
  - **Parameters**
    - `alelo` - Valor do alelo

- *imprimirAlelo*
  `public void` **imprimirAlelo**`( )`
  - **Usage**
    - Permite imprimir o alelo

- *obterAlelo*
  `public int` **obterAlelo**`( )`
  - **Usage**
    - Permite obter o valor do alelo
  - **Returns** - Valor do alelo

- *toString*
  `public String` **toString**`( )`
  - **Usage**
    - Transforma o alelo numa string
  - **Returns** - String com o alelo

2.1.2 Class **Avaliacao**

Esta classe implementa mecanismos que permitem verificar a aptidão de um cromossoma e se este é admissível para o problema.

Declaration

public class Avaliacao
extends java.lang.Object

Fields

- public static final int MAXIMIZAR
  - Esta constante identifica a maximização da função objectivo/aptidão
- public static final int MINIMIZAR
  - Esta constante identifica a minimização da função objectivo/aptidão

Constructors

- *Avaliacao*
  `public` **Avaliacao**`( )`

Methods

- *obterAptidao*
  `public static int` **obterAptidao**`( pc.dissertacao.ua.algoevo.Cromossoma` **cromossoma** `)`
  - **Usage**
    - Permite obter a aptidão de um cromossoma
  - **Parameters**
    - `cromossoma` - Cromossoma a avaliar
  - **Returns** - Aptidão de um cromossoma
- *verificarCromossomaPruferAdmissivelASCM*
  `public static Object` **verificarCromossomaPruferAdmissivelASCM**`(`
  `java.util.Vector` **cromossoma**`, pc.dissertacao.ua.grafos.Grafo` **grafo** `)`
  - **Usage**
    - Verifica se um cromossoma com a codificação de Prüfer para o problema da árvore de suporte de custo mínimo sem restrições é admissível para o problema.
  - **Parameters**
    - `cromossoma` - Vector de inteiros que representa uma sequência de Prüfer
    - `grafo` - Problema a optimizar

  - **Returns** - Um objecto com 2 posições: na primeira posição indica se o cromossoma é admissível e na segunda posição contém o custo do cromossoma

---

- *verificarCromossomaPruferAdmissivelASCMRSalto*
  `public static Object` **verificarCromossomaPruferAdmissivelASCMRSalto(** `java.util.Vector` **cromossoma,** `pc.dissertacao.ua.grafos.Grafo` **grafo,** `int` **IDVerticeRaiz,** `int` **numeroMaximoDeSaltos** **)**
  - **Usage**
    - Verifica se um cromossoma com a codificação de Prüfer para o problema da árvore de suporte de custo mínimo com restrições de salto é admissível para o problema.
  - **Parameters**
    - `cromossoma` - Vector de inteiros que representa uma sequência de Prüfer
    - `grafo` - Problema a optimizar
    - `IDVerticeRaiz` - Número que identifica o vértice raiz
    - `numeroMaximoDeSaltos` - Número máximo de saltos entre o vértice raiz e qualquer outro vértice da árvore de suporte de custo mínimo
  - **Returns** - Um objecto com 2 posições: na primeira posição indica se o cromossoma é admissível e na segunda posição contém o custo do cromossoma

2.1.3 Class Cromossoma

Esta classe implementa mecanismos que permite criar e gerir um cromossoma.

Declaration

public class Cromossoma
extends java.lang.Object

Constructors

- *Cromossoma*
  `public` **Cromossoma(** `pc.dissertacao.ua.algoevo.Cromossoma` **cromossoma** **)**
  - **Usage**
    - Cria uma instância da classe Cromossoma que é uma cópia de outro cromossoma
  - **Parameters**
    - `cromossoma` - Instância da classe Cromossoma

- *Cromossoma*
  `public` **Cromossoma(** `java.util.Vector` **cromossoma** **)**
  - **Usage**
    - Cria uma instância da classe Cromossoma
  - **Parameters**
    - `cromossoma` - Vector de alelos

- *Cromossoma*
  `public` **Cromossoma(** `java.util.Vector` **prufer,** `int` **custo** **)**

- **Usage**
  - Cria uma instância da classe Cromossoma
- **Parameters**
  - `prufer` - Vector de inteiros que representa uma sequência de Prüfer
  - `custo` -

Methods

- *definirCusto*
  `public void` **definirCusto**`( int` **custo** `)`
  - **Usage**
    - Permite definir o custo associado à árvore representada pela sequência de Prüfer
  - **Parameters**
    - `custo` - Custo do cromossoma

- *existeAlelo*
  `public boolean` **existeAlelo**`( int` **alelo** `)`
  - **Usage**
    - Permite verificar se o alelo existe no cromossoma
  - **Parameters**
    - `alelo` - Valor do alelo a comparar com os valores dos alelos do cromossoma
  - **Returns** - Verdadeiro se esse alelo existe no cromossoma

---

- *imprimirCromossoma*
  `public void` **imprimirCromossoma**`( )`
  - **Usage**
    - Permite imprimir o cromossoma

- *imprimirCromossomaECusto*
  `public void` **imprimirCromossomaECusto**`( )`
  - **Usage**
    - Permite imprimir o cromossoma e o seu custo

- *imprimirCromossomas*
  `public static void` **imprimirCromossomas**`( java.util.Vector` **cromossomas**`, java.lang.String` **texto** `)`
  - **Usage**
    - Permite imprimir uma lista de cromossomas e os seus custos
  - **Parameters**
    - `cromossomas` - Vector de cromossomas
    - `texto` - Cabeçalho para a impressão

- *obterAlelo*
  `public Alelo` **obterAlelo**`( int` **indiceAlelo** `)`
  - **Usage**

    * Permite obter um alelo do cromossoma pela sua posição
  - **Parameters**
    * `indiceAlelo` - Posição do alelo no cromossoma
  - **Returns** - Instância da classe Alelo

- *obterCromossoma*
`public Vector` **obterCromossoma**`( )`
  - **Usage**
    * Permite obter o cromossoma actual
  - **Returns** - Vector de alelos que representam o cromossoma actual

---

- *obterCromossoma2*
`public Vector` **obterCromossoma2**`( )`
  - **Usage**
    * Permite obter o cromossoma actual
  - **Returns** - Vector de inteiros que representam o cromossoma actual

---

- *obterCusto*
`public int` **obterCusto**`( )`
  - **Usage**
    * Permite obter o custo do cromossoma
  - **Returns** - Custo do cromossoma

- *obterNumeroDeAlelos*
`public int` **obterNumeroDeAlelos**`( )`
  - **Usage**
    * Permite obter o número de alelos no cromossoma
  - **Returns** - Número de alelos no cromossoma

- *toString*
`public String` **toString**`( )`
  - **Usage**
    * Transforma o cromossoma numa string
  - **Returns** - String com o cromossoma

2.1.4 Class **Cruzamento**

Esta classe implementa mecanismos de cruzamentos de cromossomas codificados por sequências de Prüfer.

Declaration

public class Cruzamento
extends java.lang.Object

Fields

- public static final int MASCARA_VARIAVEL
  - Esta constante indica que a máscara do cruzamento muda a cada operação de cruzamento
- public static final int MASCARA_FIXA_POR_GERACAO
  - Esta constante indica que a máscara do cruzamento muda a cada geração/iteração
- public static final int MASCARA_FIXA_AUTOMATICA
  - Esta constante indica que a máscara do cruzamento é constante ao longo da execução do algoritmo evolutivo e é gerado automaticamente
- public static final int MASCARA_FIXA_MANUAL
  - Esta constante indica que a máscara do cruzamento é constante ao longo da execução do algoritmo evolutivo e é definida manualmente

Constructors

- *Cruzamento*
  `public` **Cruzamento**`( )`

Methods

- *OnePointCrossoverPruferASCM*
  `public static Vector` **OnePointCrossoverPruferASCM**`( java.util.Vector` **seleccao**`, pc.dissertacao.ua.grafos.Grafo` **grafo**`, int` **tipoDeMascara** `)`
  - **Usage**
    - Permite efectuar a operação de cruzamento One Point Crossover a uma lista de cromossomas codificados por sequências de Prüfer, para o problema das árvores de suporte de custo mínimo sem restrições
  - **Parameters**
    - `seleccao` - Vector de cromossomas seleccionados
    - `grafo` - Problema a optimizar
    - `tipoDeMascara` - Tipo de máscara a usar na operação de cruzamento
  - **Returns** - Vector de cromossomas cruzados admissíveis

- *OnePointCrossoverPruferASCM*
  `public static Vector` **OnePointCrossoverPruferASCM**`( java.util.Vector` **seleccao**`, pc.dissertacao.ua.grafos.Grafo` **grafo**`, int` **tipoDeMascara**`, int` **indiceDeCorteManual** `)`
  - **Usage**
    - Permite efectuar a operação de cruzamento One Point Crossover a uma lista de cromossomas codificados por sequências de Prüfer, para o problema das árvores de suporte de custo mínimo sem restrições (com índice de corte manual)
  - **Parameters**
    - `seleccao` - Vector de cromossomas seleccionados

  - `grafo` - Problema a optimizar
  - `tipoDeMascara` - Tipo de máscara a usar na operação de cruzamento
  - `indiceDeCorteManual` - Índice de corte do cromossoma para máscara manual
- **Returns** - Vector de cromossomas cruzados admissíveis

- *OnePointCrossoverPruferASCMRSalto*
`public static Vector` **OnePointCrossoverPruferASCMRSalto(** `java.util.Vector` **seleccao,** `pc.dissertacao.ua.grafos.Grafo` **grafo,** `int` **tipoDeMascara,** `int` **IDVerticeRaiz,** `int` **numeroMaximoDeSaltos**)

- **Usage**
  - Permite efectuar a operação de cruzamento One Point Crossover a uma lista de cromossomas codificados por sequências de Prüfer, para o problema das árvores de suporte de custo mínimo com restrições de salto
- **Parameters**
  - `seleccao` - Vector de cromossomas seleccionados
  - `grafo` - Problema a optimizar
  - `tipoDeMascara` - Tipo de máscara a usar na operação de cruzamento
  - `IDVerticeRaiz` - Número que identifica o vértice raiz
  - `numeroMaximoDeSaltos` - Número máximo de saltos entre o vértice raiz e qualquer outro vértice da árvore de suporte de custo mínimo
- **Returns** - Vector de cromossomas cruzados admissíveis

- *OnePointCrossoverPruferASCMRSalto*
`public static Vector` **OnePointCrossoverPruferASCMRSalto(** `java.util.Vector` **seleccao,** `pc.dissertacao.ua.grafos.Grafo` **grafo,** `int` **tipoDeMascara,** `int` **indiceDeCorteManual,** `int` **IDVerticeRaiz,** `int` **numeroMaximoDeSaltos**)

- **Usage**
  - Permite efectuar a operação de cruzamento One Point Crossover a uma lista de cromossomas codificados por sequências de Prüfer, para o problema das árvores de suporte de custo mínimo com restrições de salto (com índice de corte manual)
- **Parameters**
  - `seleccao` - Vector de cromossomas seleccionados
  - `grafo` - Problema a optimizar
  - `tipoDeMascara` - Tipo de máscara a usar na operação de cruzamento
  - `indiceDeCorteManual` - Índice de corte do cromossoma para máscara manual
  - `IDVerticeRaiz` - Número que identifica o vértice raiz
  - `numeroMaximoDeSaltos` - Número máximo de saltos entre o vértice raiz e qualquer outro vértice da árvore de suporte de custo mínimo
- **Returns** - Vector de cromossomas cruzados admissíveis

2.1.5 Class MotorEvolutivo

Esta classe permite executar o algoritmo evolutivo durante as várias gerações/iterações requeridas. Permite também executar testes (várias execuções do algoritmo evolutivo) e guardar os resultados num ficheiro.

Declaration

public class MotorEvolutivo
extends java.lang.Object

Fields

- public static final int ASCM
  - Constante que representa o problema da árvore de suporte de custo mínimo sem restrições
- public static final int ASCMRSalto
  - Constante que representa o problema da árvore de suporte de custo mínimo com restrições de salto

Constructors

- *MotorEvolutivo*
  `public` **MotorEvolutivo**`( )`

Methods

- *configurar*
  `public void` **configurar**`( int` **tipoASCM**`, int` **numIteracoes**`, int` **dimPopulacao**`, pc.dissertacao.ua.grafos.Grafo` **grafo**`, int` **objectivo**`, int` **dimensaoTorneio**`, int` **numFinalistasTorneio**`, int` **percentagemMutacao**`, int` **numIteracoesParaRenovacaoDaPopulacao**`, int` **numIteracoesParaImprimirResultado**`, boolean` **guardarTodasIteracoes** `)`
  - **Usage**
    - Permite configurar os parâmetros do motor evolutivo.
  - **Parameters**
    - `tipoASCM` - Tipo de árvore suporte de custo mínimo a ser usada no algoritmo evolutivo (ver constantes acima)
    - `numIteracoes` - Número de iterações do algoritmo genético
    - `dimPopulacao` - Número de elementos da população do algoritmo genético
    - `grafo` - Problema a optimizar
    - `objectivo` - Permite definir se queremos minimizar ou maximizar a função objectivo/aptidão
    - `dimensaoTorneio` - Número de elementos da população que são escolhidos ao acaso
    - `numFinalistasTorneio` - Número de elementos finalistas do torneio (mínimo são 2 pais)
    - `percentagemMutacao` - Percentagem de mutação para os elementos cruzados.
    - `numIteracoesParaRenovacaoDaPopulacao` - Número de iterações para renovação da população
    - `numIteracoesParaImprimirResultado` - Número de iterações para imprimir resultado
    - `guardarTodasIteracoes` - Permite guardar valores em todas as iterações
- *configurarASCMRSalto*
  `public void` **configurarASCMRSalto**`( pc.dissertacao.ua.grafos.Grafo` **grafo**`, int` **numeroMaximoDeSaltos**`, boolean` **usarHeuristicaGeracaoPopulacao** `)`
  - **Usage**
    - Permite configurar o algoritmo evolutivo para o problema da árvore de suporte de custo mínimo com restrições de salto (o vértice raiz é o último vértice do grafo)
  - **Parameters**

  - `grafo` - Problema a optimizar
  - `numeroMaximoDeSaltos` - Número máximo de saltos entre o vértice raiz e qualquer outro vértice da árvore de suporte de custo mínimo
  - `usarHeuristicaGeracaoPopulacao` - Usar heurística para geração da população

- *configurarASCMRSalto*
  `public void` **configurarASCMRSalto**`( pc.dissertacao.ua.grafos.Grafo` **grafo**`, int` **IDVerticeRaiz**`, int` **numeroMaximoDeSaltos**`, boolean` **usarHeuristicaGeracaoPopulacao** `)`
  - **Usage**
    - Permite configurar o algoritmo evolutivo para o problema da árvore de suporte de custo mínimo com restrições de salto, com a definição manual do vértice raiz
  - **Parameters**
    - `grafo` - Problema a optimizar
    - `IDVerticeRaiz` - Número que identifica o vértice raiz
    - `numeroMaximoDeSaltos` - Número máximo de saltos entre o vértice raiz e qualquer outro vértice da árvore de suporte de custo mínimo
    - `usarHeuristicaGeracaoPopulacao` - Usar heurística para geração da população

- *configurarMascara*
  `public void` **configurarMascara**`( int` **tipoDeMascara** `)`
  - **Usage**
    - Permite definir o tipo de máscara a usar na operação de cruzamento
  - **Parameters**
    - `tipoDeMascara` - Indica o tipo de máscara a usar nas operações de cruzamento dos elementos.

- *configurarMascara*
  `public void` **configurarMascara**`( int` **tipoDeMascara**`, int` **indiceDeCorteManual** `)`
  - **Usage**
    - Permite definir o índice de corte para a máscara a usar na operação de cruzamento (deve ser usado com a máscara manual)
  - **Parameters**
    - `tipoDeMascara` - Indica o tipo de máscara a usar nas operações de cruzamento dos elementos.
    - `indiceDeCorteManual` - ndice de corte definido manualmente para a operação de cruzamento

---

- *executar*
  `public void` **executar**`( )`
  - **Usage**
    - Permite executar o algoritmo evolutivo com a configuração definida na criação do motor evolutivo

---

- *testarCSV*
  `public void` **testarCSV**`( int` **numeroDeTestes**`, java.lang.String` **nomeFicheiroResultados** `)`

  - **Usage**
    * Permite executar um determinado número de vezes um algoritmo evolutivo com as mesmas configurações e guardar os resultados de cada execução num ficheiro CSV.
  - **Parameters**
    * `numeroDeTestes` - Número de testes a efectuar
    * `nomeFicheiroResultados` - Nome do ficheiro de resultados

- *testarExportarExcel*
  `public void` **testarExportarExcel**`( int` **numeroDeTestes**`, java.lang.String` **nomeFicheiroResultados**`, java.lang.String` **nomeFolhaExcel**`, int` **IDTeste** `)`
  - **Usage**
    * Permite executar um determinado número de vezes um algoritmo evolutivo com as mesmas configurações e guardar os resultados de cada execução num ficheiro CSV.
  - **Parameters**
    * `numeroDeTestes` - Número de testes a efectuar
    * `nomeFicheiroResultados` - Nome do ficheiro de resultados
    * `nomeFolhaExcel` - Nome da folha excel para guardar os resultados
    * `IDTeste` - Indica qual o teste que está a ser efectuado

2.1.6 Class **Mutacao**

Esta classe implementa mecanismos de mutação de cromossoma codificados por sequências de Prüfer.

Declaration

public class Mutacao **extends** java.lang.Object

Constructors

- *Mutacao*
  `public` **Mutacao**`( )`

Methods

- *MutacaoPruferASCM*
  `public static Vector` **MutacaoPruferASCM**`( java.util.Vector` **seleccao**`, pc.dissertacao.ua.grafos.Grafo` **grafo**`, int` **percentagemMutacao** `)`
  - **Usage**
    * Permite efectuar a mutação de cromossomas codificados por sequências de Prüfer para o problema da árvore de suporte de custo mínimo sem restrições. É escolhido aleatoriamente um índice para sofrer a mutação.
  - **Parameters**
    * `seleccao` - Vector de cromossomas cruzados
    * `grafo` - Problema a optimizar

    * `percentagemMutacao` - (0..100) indica a percentagem de cromossomas cruzados que sofrem uma mutação
  - **Returns** - Vector de cromossomas mutados admissíveis

- *MutacaoPruferASCMRSalto*
  `public static Vector` **MutacaoPruferASCMRSalto**`( java.util.Vector` **seleccao**`, pc.dissertacao.ua.grafos.Grafo` **grafo**`, int` **percentagemMutacao**`, int` **IDVerticeRaiz**`, int` **numeroMaximoDeSaltos** `)`
  - **Usage**
    * Permite efectuar a mutação de cromossomas codificados por sequências de Prüfer para o problema da árvore de suporte de custo mínimo com restrições de salto. É escolhido aleatoriamente um índice para sofrer a mutação.
  - **Parameters**
    * `seleccao` - Vector de cromossomas cruzados
    * `grafo` - Problema a optimizar
    * `percentagemMutacao` - (0..100) indica a percentagem de cromossomas cruzados que sofrem uma mutação
    * `IDVerticeRaiz` - Número que identifica o vértice raiz
    * `numeroMaximoDeSaltos` - Número máximo de saltos entre o vértice raiz e qualquer outro vértice da árvore de suporte de custo mínimo
  - **Returns** - Vector de cromossomas mutados admissíveis

2.1.7 Class **Populacao**

Esta classe implementa mecanismos que permitem gerar uma população.

Declaration

public class Populacao **extends** java.lang.Object

Constructors

- *Populacao*
  `public` **Populacao**`( int` **tamanho**`, pc.dissertacao.ua.grafos.Grafo` **grafo** `)`
  - **Usage**
    * Permite gerar uma instância da classe população para o problema da árvore de suporte de custo mínimo sem restrições
  - **Parameters**
    * `tamanho` - Número de elementos na população
    * `grafo` - Problema a optimizar

- *Populacao*
  `public` **Populacao**`( int` **tamanho**`, pc.dissertacao.ua.grafos.Grafo` **grafo**`, int` **IDVerticeRaiz**`, int` **numeroMaximoDeSaltos**`, boolean` **usarHeuristicaGeracaoPopulacao** `)`

- **Usage**
  - Permite gerar uma instância da classe população para o problema da árvore de suporte de custo mínimo com restrições de salto
- **Parameters**
  - `tamanho` - Número de elementos na população
  - `grafo` - Problema a optimizar
  - `IDVerticeRaiz` - Número que identifica o vértice raiz
  - `numeroMaximoDeSaltos` - Número máximo de saltos entre o vértice raiz e qualquer outro vértice da árvore de suporte de custo mínimo
  - `usarHeuristicaGeracaoPopulacao` - Usar heurística para geração da população

METHODS

- *gerarPopulacaoPruferASCM*
  `public void` **gerarPopulacaoPruferASCM**`( int` **dimensao**`, pc.dissertacao.ua.grafos.Grafo` **grafo** `)`
  - **Usage**
    - Permite gerar uma população para o problema da árvore de suporte de custo mínimo sem restrições. Nota 1: A população pode ter elementos cujo valor do cromossoma é igual. Nota 2: Todos os elementos da população são admissíveis para o problema
  - **Parameters**
    - `dimensao` - Número de elementos a gerar para a população
    - `grafo` - Problema a optimizar

- *gerarPopulacaoPruferASCMRSalto*
  `public void` **gerarPopulacaoPruferASCMRSalto**`( int` **dimensao**`, pc.dissertacao.ua.grafos.Grafo` **grafo**`, int` **IDVerticeRaiz**`, int` **numeroMaximoDeSaltos**`, boolean` **usarHeuristicaGeracaoPopulacao** `)`
  - **Usage**
    - Permite gerar uma população para o problema da árvore de suporte de custo mínimo com restrições de salto.
  - **Parameters**
    - `dimensao` - Número de elementos a gerar para a população
    - `grafo` - Problema a optimizar
    - `IDVerticeRaiz` - Número que identifica o vértice raiz
    - `numeroMaximoDeSaltos` - Número máximo de saltos entre o vértice raiz e qualquer outro vértice da árvore de suporte de custo mínimo
    - `usarHeuristicaGeracaoPopulacao` - Usar heurística para geração da população

- *imprimirPopulacao*
  `public void` **imprimirPopulacao**`( )`
  - **Usage**
    - Permite imprimir a população com o respectivo custo

- *obterCromossoma*
  `public Cromossoma` **obterCromossoma**`( int` **index** `)`
  - **Usage**

    * Permite obter um elemento (cromossoma) da população através da sua posição no vector de cromossomas
  - **Parameters**
    * `index` - Posição no vector de cromossomas
  - **Returns** - Instância da classe Cromossoma

- *obterDimensaoPopulacao*
`public int` **obterDimensaoPopulacao**`( )`
  - **Usage**
    * Permite obter o número de elementos da população
  - **Returns** - Número de elementos da população

- *obterMaisApto*
`public Cromossoma` **obterMaisApto**`( int` **MINMAX** `)`
  - **Usage**
    * Permite obter o elemento mais apto da população
  - **Parameters**
    * `MINMAX` - Indica se queremos Maximizar ou Minimizar a função objectivo
  - **Returns** - Elemento mais apto da população

- *renovarPopulacaoASCM*
`public void` **renovarPopulacaoASCM**`( int` **dimensao**`,`
`pc.dissertacao.ua.grafos.Grafo` **grafo**`, int` **MINMAX** `)`
  - **Usage**
    * Permite renovar a população para o problema da árvore de suporte de custo mínimo sem restrições, no qual o melhor elemento permanece na população e os restantes são gerados.
  - **Parameters**
    * `dimensao` - Número de elementos a gerar para a população
    * `grafo` - Problema a optimizar
    * `MINMAX` - Indica se queremos Maximizar ou Minimizar a função objectivo

---

- *renovarPopulacaoASCMRSalto*
`public void` **renovarPopulacaoASCMRSalto**`( int` **dimensao**`,`
`pc.dissertacao.ua.grafos.Grafo` **grafo**`, int` **MINMAX**`, int` **IDVerticeRaiz**`, int`
numeroMaximoDeSaltos`, boolean` **usarHeuristicaGeracaoPopulacao** `)`
  - **Usage**
    * Permite renovar a população para o problema da árvore de suporte de custo mínimo com restrições de salto, no qual o melhor elemento permanece na população e os restantes são gerados.
  - **Parameters**
    * `dimensao` - Número de elementos a gerar para a população
    * `grafo` - Problema a optimizar
    * `MINMAX` - Indica se queremos Maximizar ou Minimizar a função objectivo
    * `IDVerticeRaiz` - Número que identifica o vértice raiz
    * `numeroMaximoDeSaltos` - Número máximo de saltos entre o vértice raiz e qualquer outro vértice da árvore de suporte de custo mínimo
    * `usarHeuristicaGeracaoPopulacao` - Usar heurística para renovação da população

- *substituirMenosAptos*
  `public void` **substituirMenosAptos**`( java.util.Vector` **cromossomas**`, int` **MINMAX** `)`
  - **Usage**
    - Permite substituir os elementos menos aptos da população. Nota: Os elementos menos aptos são sempre substituídos pelos novos elementos mesmo que a aptidão destes seja inferior. Isto favorece a heterogeneidade da população.
  - **Parameters**
    - `cromossomas` - Vector com os novos cromossomas
    - `MINMAX` - Indica se queremos Maximizar ou Minimizar a função objectivo

2.1.8 Class Seleccao

Esta classe implementa mecanismos que efectuam a selecção por torneio dos elementos da população.

Declaration

public class Seleccao
extends java.lang.Object

Constructors

- *Seleccao*
  `public` **Seleccao**`( )`

Methods

- *seleccaoPorTorneio*
  `public static Vector` **seleccaoPorTorneio**`( pc.dissertacao.ua.algoevo.Populacao` **populacao**`, int` **dimensaoTorneio**`, int` **numFinalistasTorneio**`, int` **MINMAX** `)`
  - **Usage**
    - Permite obter um vector de cromossomas seleccionados por torneio
  - **Parameters**
    - `populacao` - População sobre o qual se processa o torneio
    - `dimensaoTorneio` - Número de elementos a seleccionar aleatoriamente da população
    - `numFinalistasTorneio` - Número de elementos com maior aptidão que constituem os seleccionados do torneio (deve ser inferior ou igual a numFinalistasTorneio)
    - `MINMAX` - Indica se pretendemos maximizar ou minimizar a função objectivo/aptidão
  - **Returns** - Vector de cromossomas seleccionados

Capítulo 3

Package pc.dissertacao.ua.grafos

Package Contents *Page*

Classes

3.1 Classes

3.1.1 Class **Aresta**

Esta classe implementa mecanismos que definem a aresta de um grafo.

Declaration

public class Aresta
extends java.lang.Object

Constructors

- *Aresta*
  `public` **Aresta**`( int` **id** `)`
  - **Usage**
    - Cria uma instância da classe Aresta com id definido
  - **Parameters**
    - `id` - Número associado à aresta

- *Aresta*
  `public` **Aresta**`( int` **id**`, int` **custo** `)`
  - **Usage**
    - Cria uma instância da classe Aresta com id e custo definidos
  - **Parameters**
    - `id` - Número associado à aresta
    - `custo` - Custo associado à aresta

Methods

- *imprimirAresta*
  `public void` **imprimirAresta**`( )`
  - **Usage**
    - Permite imprimir a aresta

- *obterCusto*
  `public int` **obterCusto**`( )`
  - **Usage**
    - Permite obter o custo da aresta
  - **Returns** - Custo associado à aresta

- *obterID*
  `public int` **obterID**`( )`

  - **Usage**
    * Permite obter o id da aresta
  - **Returns** - Número associado à aresta

- *toString*
  `public String` **toString**`( )`
  - **Usage**
    * Transforma o id da aresta numa string
  - **Returns** - String com o id da aresta

3.1.2 Class Codificacao

Esta classe implementa mecanismos para a codificação e descodificação de sequências e cromossomas. Está implementada a codificação Prüfer.

Declaration

public class Codificacao
extends java.lang.Object

Constructors

- *Codificacao*
  `public` **Codificacao**`( )`

Methods

- *codificarCromossomaPrufer*
  `public static Cromossoma` **codificarCromossomaPrufer**`(`
  `pc.dissertacao.ua.grafos.Grafo` **arvore** `)`
  - **Usage**
    * Permite codificar uma árvore num cromossoma de Prüfer
  - **Parameters**
    * `arvore` - Árvore de suporte a codificar
  - **Returns** - Instância da classe Cromossoma que representa a sequência de Prüfer

---

- *codificarSequenciaPrufer*
  `public static Vector` **codificarSequenciaPrufer**`( pc.dissertacao.ua.grafos.Grafo`
  arvore `)`
  - **Usage**
    * Permite codificar uma árvore numa sequência de Prüfer
  - **Parameters**
    * `arvore` - Árvore de suporte a codificar
  - **Returns** - Vector de inteiros que representa a sequência de Prüfer

- *descodificarCromossomaPrufer*
  `public static Grafo` **descodificarCromossomaPrufer**`(`
  `pc.dissertacao.ua.algoevo.Cromossoma` **cromossoma** `)`
  - **Usage**
    - Permite descodificar um cromossoma de Prüfer numa árvore
  - **Parameters**
    - `cromossoma` - Instância da classe Cromossoma que representa a sequência de Prüfer
  - **Returns** - Árvore de suporte descodificada

- *descodificarSequenciaPrufer*
  `public static Grafo` **descodificarSequenciaPrufer**`( java.util.Vector` **prufer** `)`
  - **Usage**
    - Permite descodificar uma sequência de Prüfer numa árvore
  - **Parameters**
    - `prufer` - Vector de inteiros que representa a sequência de Prüfer
  - **Returns** - Árvore de suporte descodificada

- *imprimirSequenciaPrufer*
  `public static void` **imprimirSequenciaPrufer**`( java.util.Vector` **prufer** `)`
  - **Usage**
    - Permite imprimir uma sequência de Prüfer
  - **Parameters**
    - `prufer` - Sequência de Prüfer a imprimir

3.1.3 Class **Grafo**

Esta classe implementa mecanismos de criação e trabalho com grafos, através de mapas de adjacência e de incidência.

Declaration

public class Grafo
extends java.lang.Object

Fields

- public Map mapaAdjacencia
  - Variável que representa o mapa de adjacências (IDVertice, Set)
- public Map mapaIncidencia
  - Variável que representa o mapa de incidências (IDAresta, ParDeVertices)
- public SortedMap vertices

  - Variável que representa o mapa de vértices (IDVertice, Vertice)

- public SortedMap arestas
  - Variável que representa o mapa de arestas (IDAresta, Aresta)

CONSTRUCTORS

- *Grafo*
  `public` **Grafo**`( )`
  - **Usage**
    * Cria uma instância da classe Grafo

- *Grafo*
  `public` **Grafo**`( java.util.Map` **mapaAdjacencia**`, java.util.Map` **mapaIncidencia**`, java.util.SortedMap` **vertices**`, java.util.SortedMap` **arestas** `)`
  - **Usage**
    * Cria uma instância da classe Grafo com parâmetros explícitos
  - **Parameters**
    * `mapaAdjacencia` - Mapa com as adjacências vértice-aresta (IDVertice, Set)
    * `mapaIncidencia` - Mapa com as incidências aresta-vértices (IDAresta, ParDeVertices)
    * `vertices` - Mapa de vértices (IDVertice, Vertice)
    * `arestas` - Mapa de arestas (IDAresta, Aresta)

METHODS

- *actualizarSaltos*
  `public void` **actualizarSaltos**`( int` **IDVerticeRaiz** `)`
  - **Usage**
    * Permite actualizar os saltos dos vértices de uma árvore de suporte de custo mínimo. (Deve apenas ser usado com instâncias desta classe que sejam árvores)
  - **Parameters**
    * `IDVerticeRaiz` - Número que identifica o vértice raiz

- *adicionarAresta*
  `public boolean` **adicionarAresta**`( int` **IDVertice1**`, int` **IDVertice2** `)`
  - **Usage**
    * Permite adicionar uma aresta ao grafo
  - **Parameters**
    * `IDVertice1` - Número que identifica um vértice extremo
    * `IDVertice2` - Número que identifica outro vértice extremo
  - **Returns** - Verdadeiro se a aresta foi adicionada ao grafo

---

- *adicionarAresta*
  `public boolean` **adicionarAresta**`( int` **IDVertice1**`, int` **IDVertice2**`, int` **custo** `)`
  - **Usage**
    * Permite adicionar uma aresta ao grafo

  - **Parameters**
    * `IDVertice1` - Número que identifica um vértice extremo
    * `IDVertice2` - Número que identifica outro vértice extremo
    * `custo` - Custo associado a aresta
  - **Returns** - Verdadeiro se a aresta foi adicionada ao grafo

---

- *adicionarVertice*
`public boolean` **adicionarVertice**`( int` **IDVertice** `)`
  - **Usage**
    * Permite adicionar um vértice ao grafo
  - **Parameters**
    * `IDVertice` - Número do vértice a adicionar
  - **Returns** - Verdadeiro se o vértice foi adicionado ao grafo

---

- *clonarGrafo*
`public Grafo` **clonarGrafo**`( )`
  - **Usage**
    * Permite obter uma cópia do grafo actual
  - **Returns** - Uma instância da classe Grafo com os mesmos valores do grafo actual

---

- *existeAresta*
`public boolean` **existeAresta**`( pc.dissertacao.ua.grafos.Aresta` **aresta** `)`
  - **Usage**
    * Permite verificar se uma aresta existe no grafo
  - **Parameters**
    * `aresta` - Instância da classe Aresta a verificar
  - **Returns** - Verdadeiro se essa instância existe no grafo

---

- *existeAresta*
`public boolean` **existeAresta**`( int` **IDAresta** `)`
  - **Usage**
    * Permite verificar se uma aresta existe no grafo
  - **Parameters**
    * `IDAresta` - Número que identifica a aresta
  - **Returns** - Verdadeiro se existe no grafo uma aresta com esse id

---

- *existeAresta*
`public boolean` **existeAresta**`( int` **IDVertice1**`, int` **IDVertice2** `)`
  - **Usage**
    * Permite verificar se uma aresta existe no grafo
  - **Parameters**
    * `IDVertice1` - Número que identifica um vértice extremo
    * `IDVertice2` - Número que identifica outro vértice extremo
  - **Returns** - Verdadeiro se existe no grafo uma aresta com esses vértices extremos

---

- *existeVertice*
  `public boolean` **existeVertice**`( int` **IDVertice** `)`
  - **Usage**
    - Permite verificar se um vértice existe no grafo
  - **Parameters**
    - `IDVertice` - Número que identifica o vértice
  - **Returns** - Verdadeiro se existe no grafo um vértice com esse id

---

- *existeVertice*
  `public boolean` **existeVertice**`( pc.dissertacao.ua.grafos.Vertice` **vertice** `)`
  - **Usage**
    - Permite verificar se um vértice existe no grafo
  - **Parameters**
    - `vertice` - Instância da classe Vertice a verificar
  - **Returns** - Verdadeiro se essa instância existe no grafo

---

- *gerarCromossomaPruferASCM*
  `public Cromossoma` **gerarCromossomaPruferASCM**`( )`
  - **Usage**
    - Permite gerar uma instância de Cromossoma para a árvore de suporte de custo mínimo com restrições que é admissível para o grafo actual
  - **Returns** - Uma instância de Cromossoma que é admissível para o grafo actual

---

- *gerarCromossomaPruferASCMRSalto*
  `public Cromossoma` **gerarCromossomaPruferASCMRSalto**`( int` **IDVerticeRaiz**`, int` **numeroMaximoDeSaltos** `)`
  - **Usage**
    - Permite gerar aleatoriamente uma instância de Cromossoma para a árvore de suporte de custo mínimo com restrições de salto que é admissível para o grafo actual
  - **Parameters**
    - `IDVerticeRaiz` - Número que identifica o vértice raiz
    - `numeroMaximoDeSaltos` - Número máximo de saltos entre o vértice raiz e qualquer outro vértice da árvore de suporte de custo mínimo
  - **Returns** - Uma instância de Cromossoma que é admissível para o grafo actual

---

- *gerarCromossomaPruferASCMRSaltoHeuristica*
  `public Cromossoma` **gerarCromossomaPruferASCMRSaltoHeuristica**`( int` **IDVerticeRaiz**`, int` **numeroMaximoDeSaltos** `)`
  - **Usage**
    - Permite gerar uma instância de Cromossoma para a árvore de suporte de custo mínimo com restrições de salto que é admissível para o grafo actual
  - **Parameters**
    - `IDVerticeRaiz` - Número que identifica o vértice raiz
    - `numeroMaximoDeSaltos` - Número máximo de saltos entre o vértice raiz e qualquer outro vértice da árvore de suporte de custo mínimo
  - **Returns** - Uma instância de Cromossoma que é admissível para o grafo actual

---

- *imprimirAdjacencias*
  `public void` **imprimirAdjacencias**`( )`
  - **Usage**
    - Permite imprimir a lista de adjacências do grafo

- *imprimirArestas*
  `public void` **imprimirArestas**`( )`
  - **Usage**
    - Permite imprimir a lista das arestas do grafo

- *imprimirIncidencias*
  `public void` **imprimirIncidencias**`( )`
  - **Usage**
    - Permite imprimir a lista de incidências do grafo

- *imprimirVertices*
  `public void` **imprimirVertices**`( )`
  - **Usage**
    - Permite imprimir a lista dos vértices do grafo

- *imprimirVerticesESaltos*
  `public void` **imprimirVerticesESaltos**`( )`
  - **Usage**
    - Permite imprimir a lista dos vértices (e saltos) do grafo

---

- *obterAresta*
  `public Aresta` **obterAresta**`( int` **IDVertice1**`, int` **IDVertice2** `)`
  - **Usage**
    - Permite obter uma aresta existente no grafo
  - **Parameters**
    - `IDVertice1` - Número que identifica um vértice extremo
    - `IDVertice2` - Número que identifica outro vértice extremo
  - **Returns** - Null se a aresta não existe, caso contrário retorna uma instância da classe Aresta

---

- *obterArestas*
  `public Collection` **obterArestas**`( )`
  - **Usage**
    - Permite obter a lista de aresta que estão no grafo
  - **Returns** - Lista de aresta que estão no grafo

- *obterIDVerticeVizinho*
  `public int` **obterIDVerticeVizinho**`( int` **IDVertice** `)`
  - **Usage**
    - Permite obter o id de um vértice vizinho de um determinado vértice (Usar sempre em conjunto com obterNumeroDeVerticesVizinhos)

  - **Parameters**
    - `IDVertice` - Número que identifica o vértice
  - **Returns** - Id de um vértice vizinho desse vértice

- *obterNumeroDeArestas*
  `public int` **obterNumeroDeArestas**`( )`
  - **Usage**
    - Permite obter o número de arestas no grafo
  - **Returns** - Número de arestas no grafo

- *obterNumeroDeVertices*
  `public int` **obterNumeroDeVertices**`( )`
  - **Usage**
    - Permite obter o número de vértices no grafo
  - **Returns** - Número de vértices no grafo

- *obterNumeroDeVerticesVizinhos*
  `public int` **obterNumeroDeVerticesVizinhos**`( int` **IDVertice** `)`
  - **Usage**
    - Permite obter o número de vértices vizinhos de um determinado vértice
  - **Parameters**
    - `IDVertice` - Número que identifica o vértice
  - **Returns** - Número de vértices vizinhos desse vértice

- *obterParDeVertices*
  `public Collection` **obterParDeVertices**`( )`
  - **Usage**
    - Permite obter os pares de vértices do mapa de incidência
  - **Returns** - Pares de vértices do mapa de incidência

- *obterVertices*
  `public Collection` **obterVertices**`( )`
  - **Usage**
    - Permite obter a lista de vértices que estão no grafo
  - **Returns** - Lista de vértices que estão no grafo

- *removerTodosVertices*
  `public void` **removerTodosVertices**`( )`
  - **Usage**
    - Permite remover todos os vértices do grafo

- *removerVertice*
  `public boolean` **removerVertice**`( int` **IDVertice** `)`
  - **Usage**
    - Permite remover um vértice do grafo
  - **Parameters**
    - `IDVertice` - Número que identifica o vértice
  - **Returns** - Verdadeiro se o vértice foi removido do grafo

3.1.4 Class **Grafo.ParDeVertices**

Esta classe permite agrupar 2 vértices para armazenar no mapa de incidências

Declaration

public class Grafo.ParDeVertices
extends java.lang.Object

Constructors

- *Grafo.ParDeVertices*
  `public` **Grafo.ParDeVertices**`( int` **IDVertice1**`, int` **IDVertice2** `)`
  - **Usage**
    - Cria uma nova instância da classe
  - **Parameters**
    - `IDVertice1` - Número que identifica um vértice
    - `IDVertice2` - Número que identifica outro vértice

Methods

- *imprimirParDeVertices*
  `public void` **imprimirParDeVertices**`( )`
  - **Usage**
    - Permite imprimir o par de vértices

- *obterIDVertice1*
  `public int` **obterIDVertice1**`( )`
  - **Usage**
    - Permite obter o id de um vértice pertencente ao par de vértices
  - **Returns** - Id de um vértice pertencente ao par de vértices

---

- *obterIDVertice2*
  `public int` **obterIDVertice2**`( )`
  - **Usage**
    - Permite obter o id de outro vértice pertencente ao par de vértices
  - **Returns** - Id de outro vértice pertencente ao par de vértices

---

- *toString*
  `public String` **toString**`( )`
  - **Usage**
    - Transforma o par de vértices numa string
  - **Returns** - String com os ids dos vértices

3.1.5 Class **Vertice**

Esta classe implementa mecanismos que definem o vértice de um grafo.

Declaration

public class Vertice
extends java.lang.Object

Constructors

- *Vertice*
  `public` **Vertice**`( int` **id** `)`
  - **Usage**
    - Cria uma instância da classe Vertice com id definido
  - **Parameters**
    - `id` - Número associado ao vértice

Methods

- *definirSalto*
  `public boolean` **definirSalto**`( int` **salto** `)`
  - **Usage**
    - Permite definir o salto associado a este vértice em relação a um vértice raiz
  - **Parameters**
    - `salto` - Número de saltos até ao vértice raiz
  - **Returns** - Indica se o número de saltos foi adicionado com sucesso ao vértice

---

- *imprimirVertice*
  `public void` **imprimirVertice**`( )`
  - **Usage**
    - Permite imprimir o vértice

- *imprimirVerticeESalto*
  `public void` **imprimirVerticeESalto**`( )`
  - **Usage**
    - Permite imprimir o vértice e salto

- *obterID*
  `public int` **obterID**`( )`
  - **Usage**
    - Permite obter o id do vértice
  - **Returns** - Número associado ao vértice

- *obterSalto*
  `public int` **obterSalto**`( )`
  - **Usage**
    - Permite obter o número de saltos até ao vértice raiz
  - **Returns** - Número de saltos até ao vértice raiz

- *toString*
  `public String` **toString**`( )`
  - **Usage**
    - Transforma o id do vértice numa string
  - **Returns** - String com o id do vértice

Apêndice C

Documentação da Implementação do Algoritmo Genético em Java para a Codificação por Sequências de Arestas

Capítulo 1

Package pc.dissertacao.ua.runtime

Package Contents *Page*

Classes

1.1 Classes

1.1.1 Class **Consola**

Class main do projecto.

Declaration

public class Consola **extends** java.lang.Object

Constructors

- *Consola*
  `public` **Consola**`( )`

Methods

- *main*
  `public static void` **main**`( java.lang.String [] args )`
  - **Parameters**
    - `args` - Argumentos passados pela linha de comandos

1.1.2 Class **Ficheiro**

Esta classe implementa mecanismos de leitura de ficheiros com matrizes de adjacência para a construção/inicialização dos problemas, e permite guardar os resultados dos testes efectuados.

Declaration

public class Ficheiro **extends** java.lang.Object

Fields

- public static final int PARAMETRO_CUSTO
  - Constante que indica que o parâmetro de trabalho é um parâmetro de custo

Constructors

- *Ficheiro*
  `public` **Ficheiro**`( )`

METHODS

- *escreverResultadosCSV*
  `public static boolean` **escreverResultadosCSV**`( java.lang.String` **nomeDoFicheiro**`, java.lang.String` **resultados** `)`
  - **Usage**
    - Permite guardar os dados de uma sessão de testes num ficheiro separado por ponto-e-virgulas ';'. Os dados guardados são: Número de cada teste, Tempo de cada teste, Número de cruzamentos com sucesso de cada teste, Número de cruzamentos sem sucesso de cada teste, Número de mutações com sucesso de cada teste, Número de mutações sem sucesso de cada teste, O elemento mais apto de cada teste, O custo do elemento mais apto de cada teste
  - **Parameters**
    - `nomeDoFicheiro` - Nome do ficheiro a guardar os dados do teste
    - `resultados` - Resultados do teste
  - **Returns** - Indica se os dados foram guardados com sucesso no ficheiro

---

- *escreverResultadosExcel*
  `public static boolean` **escreverResultadosExcel**`( java.lang.String` **nomeDoFicheiro**`, java.lang.String` **nomeFolhaExcel**`, int` **numLinha**`, int` **tipoASCM**`, boolean` **utilizaHeuristica**`, boolean` **ultimoTeste**`, int` **objectivo**`, int` **IDTeste** `)`
  - **Usage**
    - Permite guardar os dados de uma sessão de testes num ficheiro com formato excel.
  - **Parameters**
    - `nomeDoFicheiro` - Nome do ficheiro a guardar os dados do teste
    - `nomeFolhaExcel` - Nome da folha de excel a guardar
    - `numLinha` - Número de execução do teste actual
    - `tipoASCM` - Tipo de árvore de suporte de custo mínimo
    - `utilizaHeuristica` - Indica se é utilizada a heurística para geração da população
    - `ultimoTeste` - Indica se é o último teste
    - `objectivo` - Indica qual o objectivo do teste (minimização ou maximização)
    - `IDTeste` - Indica qual o teste que está a ser guardado
  - **Returns** - Indica se os dados foram guardados com sucesso no ficheiro excel

---

- *escreverResultadosIter*
  `public static boolean` **escreverResultadosIter**`( )`
  - **Usage**
    - Permite guardar os dados de cada iteração num num ficheiro separado por ponto-e-virgulas ';'. Os dados guardados são: Número de cada iteração, O custo do elemento mais apto de cada iteração, Tempo de cada iteração
  - **Returns** - Indica se os dados foram guardados com sucesso no ficheiro

---

- *escreverResumoExcel*
  `public static boolean` **escreverResumoExcel**`( java.lang.String` **nomeDoFicheiro**`, int` **numRepeticoes** `)`
  - **Usage**
    - Permite gerar uma folha excel com o resumo dos testes efectuados

  - **Parameters**
    - `nomeDoFicheiro` - Nome do ficheiro a guardar os dados do teste
    - `numRepeticoes` - Número total de execução de cada teste
  - **Returns** - Indica se o resumo foi guardado com sucesso no ficheiro excel

---

- *lerFicheiroMatrizDeAdjacenciaTriangularSuperior*
  `public static boolean` **lerFicheiroMatrizDeAdjacenciaTriangularSuperior**`( int` **tipoParametro**`, java.lang.String` **enderecoDoFicheiro**`, pc.dissertacao.ua.grafos.Grafo` **grafo**`, boolean` **debug**`, int` **numLinhasPorLinhaDaMatriz**`, int` **limiteDeVertices** `)`

  - **Usage**
    - Permite ler um ficheiro com uma matriz de adjacência no qual os dados relevantes estão na parte triangular superior da matriz
  - **Parameters**
    - `tipoParametro` - Indica o tipo de parâmetro a ler (ver constantes acima)
    - `enderecoDoFicheiro` - Endereço do ficheiro a ler
    - `grafo` - Grafo para adicionar os vértices e arestas lidas
    - `debug` - Escreve na consola todos os outputs (caso true)
    - `numLinhasPorLinhaDaMatriz` - Indica quantas linhas no ficheiro correspondem a uma linha na matriz de adjacência
    - `limiteDeVertices` - Indica se o ficheiro é lido na totalidade (0) ou se é forçada a obtenção de um grafo não completo com um determinado número de vértices
  - **Returns** - Indica se o ficheiro foi lido com sucesso

1.1.3 Class **Sondagem**

Esta classe permite registar os dados de um teste (uma execução completa do algoritmo evolutivo) para posterior armazenamento num ficheiro.

Declaration

public class Sondagem **extends** java.lang.Object

Fields

- public static int NUMERO_DO_TESTE
  - Variável que indica o número do teste
- public static long TEMPO_DO_TESTE
  - Variável que indica o tempo do teste
- public static long NUM_CRUZAMENTOS_COM_SUCESSO
  - Variável que indica o número de cruzamentos com sucesso no teste
- public static long NUM_CRUZAMENTOS_SEM_SUCESSO

  - Variável que indica o número de cruzamentos sem sucesso no teste
- public static long NUM_MUTACOES
  - Variável que indica o número de mutações no teste
- public static long NUM_MUTACOES_COM_SUCESSO
  - Variável que indica o número de mutações com sucesso no teste
- public static long NUM_MUTACOES_SEM_SUCESSO
  - Variável que indica o número de mutações sem sucesso no teste
- public static String ELEMENTO_MAIS_APTO
  - Variável que indica o elemento mais apto do teste
- public static int CUSTO_ELEMENTO_MAIS_APTO
  - Variável que indica o custo do elemento mais apto do teste
- public static boolean MUTADO
  - Variável que indica se o cromossoma actual foi mutado
- public static Vector ELEMENTO_MAIS_APTO_ITER
  - Variável que guarda o custo do elemento mais apto do teste por iteração
- public static Vector TEMPO_ITER
  - Variável que guarda o tempo de cada iteração

CONSTRUCTORS

- *Sondagem*
  `public` **Sondagem**`( )`

METHODS

- *imprimirSondagem*
  `public static void` **imprimirSondagem**`( )`
  - **Usage**
    * Permite imprimir os dados recolhidos durante a execução do algoritmo evolutivo
- *obterSondagem*
  `public static String` **obterSondagem**`( )`
  - **Usage**
    * Permite obter os resultados de uma execução do algoritmo evolutivo no formato usado para armazenamento no ficheiro.
  - **Returns** - resultados de uma execução do algoritmo evolutivo
- *repor*
  `public static void` **repor**`( )`
  - **Usage**
    * Repõe os valores da sondagem a zero. Deve ser chamado após cada execução completa do algoritmo evolutivo.

1.1.4 Class **Testes**

Classe que permite criar conjuntos de teste para execução em batch (bloco)

Declaration

public class Testes
extends java.lang.Object

Constructors

- *Testes*
  `public` **Testes**`( )`

Methods

- *resumoExcel*
  `public static void` **resumoExcel**`( java.lang.String` **nomeDoFicheiro**`, int` **numRepeticoes** `)`
  - **Usage**
    - Permite gerar uma folha excel com o resumo dos testes efectuados
  - **Parameters**
    - `nomeDoFicheiro` - Nome do ficheiro a guardar os dados do teste
    - `numRepeticoes` - Número total de execução de cada teste

- *Teste1*
  `public static void` **Teste1**`( java.lang.String` **nomeDoFicheiro**`, int` **numLinhasPorLinhaDaMatriz**`, boolean` **usarHeuristicaGeracaoPopulacao**`, int` **numTestes** `)`
  - **Usage**
    - Permite executar o conjunto de testes associados ao Teste 1
  - **Parameters**
    - `nomeDoFicheiro` - Nome do ficheiro de dados
    - `numLinhasPorLinhaDaMatriz` - Indica quantas linhas no ficheiro correspondem a uma linha na matriz de adjacência
    - `usarHeuristicaGeracaoPopulacao` - Usar heurística para geração da população
    - `numTestes` - Número de testes a efectuar

- *Teste10*
  `public static void` **Teste10**`( java.lang.String` **nomeDoFicheiro**`, int` **numLinhasPorLinhaDaMatriz**`, boolean` **usarHeuristicaGeracaoPopulacao**`, int` **numTestes** `)`
  - **Usage**
    - Permite executar o conjunto de testes associados ao Teste 10
  - **Parameters**

  - * `nomeDoFicheiro` - Nome do ficheiro de dados
  - * `numLinhasPorLinhaDaMatriz` - Indica quantas linhas no ficheiro correspondem a uma linha na matriz de adjacência
  - * `usarHeuristicaGeracaoPopulacao` - Usar heurística para geração da população
  - * `numTestes` - Número de testes a efectuar

- *Teste11*
  `public static void` **Teste11**`( java.lang.String` **nomeDoFicheiro**`, int` **numLinhasPorLinhaDaMatriz**`, boolean` **usarHeuristicaGeracaoPopulacao**`, int` **numTestes** `)`
  - **Usage**
    - * Permite executar o conjunto de testes associados ao Teste 11
  - **Parameters**
    - * `nomeDoFicheiro` - Nome do ficheiro de dados
    - * `numLinhasPorLinhaDaMatriz` - Indica quantas linhas no ficheiro correspondem a uma linha na matriz de adjacência
    - * `usarHeuristicaGeracaoPopulacao` - Usar heurística para geração da população
    - * `numTestes` - Número de testes a efectuar

- *Teste12*
  `public static void` **Teste12**`( java.lang.String` **nomeDoFicheiro**`, int` **numLinhasPorLinhaDaMatriz**`, boolean` **usarHeuristicaGeracaoPopulacao**`, int` **numTestes** `)`
  - **Usage**
    - * Permite executar o conjunto de testes associados ao Teste 12
  - **Parameters**
    - * `nomeDoFicheiro` - Nome do ficheiro de dados
    - * `numLinhasPorLinhaDaMatriz` - Indica quantas linhas no ficheiro correspondem a uma linha na matriz de adjacência
    - * `usarHeuristicaGeracaoPopulacao` - Usar heurística para geração da população
    - * `numTestes` - Número de testes a efectuar

- *Teste2*
  `public static void` **Teste2**`( java.lang.String` **nomeDoFicheiro**`, int` **numLinhasPorLinhaDaMatriz**`, boolean` **usarHeuristicaGeracaoPopulacao**`, int` **numTestes** `)`
  - **Usage**
    - * Permite executar o conjunto de testes associados ao Teste 2
  - **Parameters**
    - * `nomeDoFicheiro` - Nome do ficheiro de dados
    - * `numLinhasPorLinhaDaMatriz` - Indica quantas linhas no ficheiro correspondem a uma linha na matriz de adjacência
    - * `usarHeuristicaGeracaoPopulacao` - Usar heurística para geração da população
    - * `numTestes` - Número de testes a efectuar

- *Teste3*
  `public static void` **Teste3**`( java.lang.String` **nomeDoFicheiro**`, int` **numLinhasPorLinhaDaMatriz**`, boolean` **usarHeuristicaGeracaoPopulacao**`, int` **numTestes** `)`
  - **Usage**

    * Permite executar o conjunto de testes associados ao Teste 3
  - **Parameters**
    * `nomeDoFicheiro` - Nome do ficheiro de dados
    * `numLinhasPorLinhaDaMatriz` - Indica quantas linhas no ficheiro correspondem a uma linha na matriz de adjacência
    * `usarHeuristicaGeracaoPopulacao` - Usar heurística para geração da população
    * `numTestes` - Número de testes a efectuar

- *Teste4*
  `public static void` **Teste4**`( java.lang.String` **nomeDoFicheiro**`, int` **numLinhasPorLinhaDaMatriz**`, boolean` **usarHeuristicaGeracaoPopulacao**`, int` **numTestes** `)`
  - **Usage**
    * Permite executar o conjunto de testes associados ao Teste 4
  - **Parameters**
    * `nomeDoFicheiro` - Nome do ficheiro de dados
    * `numLinhasPorLinhaDaMatriz` - Indica quantas linhas no ficheiro correspondem a uma linha na matriz de adjacência
    * `usarHeuristicaGeracaoPopulacao` - Usar heurística para geração da população
    * `numTestes` - Número de testes a efectuar

- *Teste5*
  `public static void` **Teste5**`( java.lang.String` **nomeDoFicheiro**`, int` **numLinhasPorLinhaDaMatriz**`, boolean` **usarHeuristicaGeracaoPopulacao**`, int` **numTestes** `)`
  - **Usage**
    * Permite executar o conjunto de testes associados ao Teste 5
  - **Parameters**
    * `nomeDoFicheiro` - Nome do ficheiro de dados
    * `numLinhasPorLinhaDaMatriz` - Indica quantas linhas no ficheiro correspondem a uma linha na matriz de adjacência
    * `usarHeuristicaGeracaoPopulacao` - Usar heurística para geração da população
    * `numTestes` - Número de testes a efectuar

- *Teste6*
  `public static void` **Teste6**`( java.lang.String` **nomeDoFicheiro**`, int` **numLinhasPorLinhaDaMatriz**`, boolean` **usarHeuristicaGeracaoPopulacao**`, int` **numTestes** `)`
  - **Usage**
    * Permite executar o conjunto de testes associados ao Teste 6
  - **Parameters**
    * `nomeDoFicheiro` - Nome do ficheiro de dados
    * `numLinhasPorLinhaDaMatriz` - Indica quantas linhas no ficheiro correspondem a uma linha na matriz de adjacência
    * `usarHeuristicaGeracaoPopulacao` - Usar heurística para geração da população
    * `numTestes` - Número de testes a efectuar

- *Teste7*
  `public static void` **Teste7**`( java.lang.String` **nomeDoFicheiro**`, int` **numLinhasPorLinhaDaMatriz**`, boolean` **usarHeuristicaGeracaoPopulacao**`, int` **numTestes** `)`

  - **Usage**
    * Permite executar o conjunto de testes associados ao Teste 7
  - **Parameters**
    * `nomeDoFicheiro` - Nome do ficheiro de dados
    * `numLinhasPorLinhaDaMatriz` - Indica quantas linhas no ficheiro correspondem a uma linha na matriz de adjacência
    * `usarHeuristicaGeracaoPopulacao` - Usar heurística para geração da população
    * `numTestes` - Número de testes a efectuar

- *Teste8*
`public static void` **Teste8**`( java.lang.String` **nomeDoFicheiro**`, int` **numLinhasPorLinhaDaMatriz**`, boolean` **usarHeuristicaGeracaoPopulacao**`, int` **numTestes** `)`

  - **Usage**
    * Permite executar o conjunto de testes associados ao Teste 8
  - **Parameters**
    * `nomeDoFicheiro` - Nome do ficheiro de dados
    * `numLinhasPorLinhaDaMatriz` - Indica quantas linhas no ficheiro correspondem a uma linha na matriz de adjacência
    * `usarHeuristicaGeracaoPopulacao` - Usar heurística para geração da população
    * `numTestes` - Número de testes a efectuar

- *Teste9*
`public static void` **Teste9**`( java.lang.String` **nomeDoFicheiro**`, int` **numLinhasPorLinhaDaMatriz**`, boolean` **usarHeuristicaGeracaoPopulacao**`, int` **numTestes** `)`

  - **Usage**
    * Permite executar o conjunto de testes associados ao Teste 9
  - **Parameters**
    * `nomeDoFicheiro` - Nome do ficheiro de dados
    * `numLinhasPorLinhaDaMatriz` - Indica quantas linhas no ficheiro correspondem a uma linha na matriz de adjacência
    * `usarHeuristicaGeracaoPopulacao` - Usar heurística para geração da população
    * `numTestes` - Número de testes a efectuar

Capítulo 2

Package pc.dissertacao.ua.algoevo

Package Contents *Page*

Classes

2.1 Classes

2.1.1 Class **Alelo**

Esta classe implementa mecanismos que definem um alelo de um cromossoma

Declaration

public class Alelo **extends** java.lang.Object

Constructors

- *Alelo*
  `public` **Alelo(** `pc.dissertacao.ua.grafos.ParDeVertices` **alelo)**
  - **Usage**
    - Cria uma instância da classe Alelo
  - **Parameters**
    - `alelo` - Valor do alelo

Methods

- *definirAlelo*
  `public void` **definirAlelo(** `pc.dissertacao.ua.grafos.ParDeVertices` **alelo)**
  - **Usage**
    - Permite definir o valor do alelo
  - **Parameters**
    - `alelo` - Valor do alelo
- *imprimirAlelo*
  `public void` **imprimirAlelo(** **)**
  - **Usage**
    - Permite imprimir o alelo
- *obterAlelo*
  `public ParDeVertices` **obterAlelo(** **)**
  - **Usage**
    - Permite obter o valor do alelo
  - **Returns** - Valor do alelo
- *toString*
  `public String` **toString(** **)**
  - **Usage**
    - Transforma o alelo numa string
  - **Returns** - String com o alelo

2.1.2 Class **Avaliacao**

Esta classe implementa mecanismos que permitem verificar a aptidão de um cromossoma e se este é admissível para o problema.

Declaration

public class Avaliacao
extends java.lang.Object

Fields

- public static final int MAXIMIZAR
  - Esta constante identifica a maximização da função objectivo/aptidão
- public static final int MINIMIZAR
  - Esta constante identifica a minimização da função objectivo/aptidão

Constructors

- *Avaliacao*
  `public` **Avaliacao**`( )`

Methods

- *obterAptidao*
  `public static int` **obterAptidao**`( pc.dissertacao.ua.algoevo.Cromossoma` **cromossoma** `)`
  - **Usage**
    - Permite obter a aptidão de um cromossoma
  - **Parameters**
    - `cromossoma` - Cromossoma a avaliar
  - **Returns** - Aptidão de um cromossoma
- *verificarCromossomaArestasAdmissivelASCM*
  `public static Object` **verificarCromossomaArestasAdmissivelASCM**`(`
  `java.util.Vector` **cromossoma**`, pc.dissertacao.ua.grafos.Grafo` **grafo** `)`
  - **Usage**
    - Verifica se um cromossoma com a codificação de Arestas para o problema da árvore de suporte de custo mínimo sem restrições é admissível para o problema.
  - **Parameters**
    - `cromossoma` - Vector de inteiros que representa uma sequência de Arestas
    - `grafo` - Problema a optimizar

  - **Returns** - Um objecto com 2 posições: na primeira posição indica se o cromossoma é admissível e na segunda posição contém o custo do cromossoma

---

- *verificarCromossomaArestasAdmissivelASCMRSalto*
  `public static Object` **verificarCromossomaArestasAdmissivelASCMRSalto(** `java.util.Vector` **cromossoma,** `pc.dissertacao.ua.grafos.Grafo` **grafo,** `int` **IDVerticeRaiz,** `int` **numeroMaximoDeSaltos**)
  - **Usage**
    - Verifica se um cromossoma com a codificação de Arestas para o problema da árvore de suporte de custo mínimo com restrições de salto é admissível para o problema.
  - **Parameters**
    - `cromossoma` - Vector de inteiros que representa uma sequência de Arestas
    - `grafo` - Problema a optimizar
    - `IDVerticeRaiz` - Número que identifica o vértice raiz
    - `numeroMaximoDeSaltos` - Número máximo de saltos entre o vértice raiz e qualquer outro vértice da árvore de suporte de custo mínimo
  - **Returns** - Um objecto com 2 posições: na primeira posição indica se o cromossoma é admissível e na segunda posição contém o custo do cromossoma

2.1.3 Class **Cromossoma**

Esta classe implementa mecanismos que permite criar e gerir um cromossoma.

Declaration

public class Cromossoma **extends** java.lang.Object

Constructors

- *Cromossoma*
  `public` **Cromossoma(** `pc.dissertacao.ua.algoevo.Cromossoma` **cromossoma**)
  - **Usage**
    - Cria uma instância da classe Cromossoma que é uma cópia de outro cromossoma
  - **Parameters**
    - `cromossoma` - Instância da classe Cromossoma

- *Cromossoma*
  `public` **Cromossoma(** `java.util.Vector` **cromossoma**)
  - **Usage**
    - Cria uma instância da classe Cromossoma
  - **Parameters**
    - `cromossoma` - Vector de alelos

- *Cromossoma*
  `public` **Cromossoma(** `java.util.Vector` **seqArestas,** `int` **custo**)

  - **Usage**
    * Cria uma instância da classe Cromossoma
  - **Parameters**
    * `seqArestas` - Vector com pares de vértices
    * `custo` - Custo do cromossoma

Methods

- *definirCusto*
  `public void` **definirCusto**`( int` **custo** `)`
  - **Usage**
    * Permite definir o custo associado à árvore representada pela sequência de Arestas
  - **Parameters**
    * `custo` - Custo do cromossoma

- *existeAlelo*
  `public boolean` **existeAlelo**`( pc.dissertacao.ua.grafos.ParDeVertices` **alelo** `)`
  - **Usage**
    * Permite verificar se o alelo existe no cromossoma
  - **Parameters**
    * `alelo` - Valor do alelo a comparar com os valores dos alelos do cromossoma
  - **Returns** - Verdadeiro se esse alelo existe no cromossoma

---

- *imprimirCromossoma*
  `public void` **imprimirCromossoma**`( )`
  - **Usage**
    * Permite imprimir o cromossoma

- *imprimirCromossomaECusto*
  `public void` **imprimirCromossomaECusto**`( )`
  - **Usage**
    * Permite imprimir o cromossoma e o seu custo

- *imprimirCromossomas*
  `public static void` **imprimirCromossomas**`( java.util.Vector` **cromossomas**`, java.lang.String` **texto** `)`
  - **Usage**
    * Permite imprimir uma lista de cromossomas e os seus custos
  - **Parameters**
    * `cromossomas` - Vector de cromossomas
    * `texto` - Cabeçalho para a impressão

- *obterAlelo*
  `public Alelo` **obterAlelo**`( int` **indiceAlelo** `)`
  - **Usage**

    * Permite obter um alelo do cromossoma pela sua posição
  - **Parameters**
    * `indiceAlelo` - Posição do alelo no cromossoma
  - **Returns** - Instância da classe Alelo

- *obterCromossoma*
`public Vector` **obterCromossoma**`( )`
  - **Usage**
    * Permite obter o cromossoma actual
  - **Returns** - Vector de alelos que representam o cromossoma actual

---

- *obterCromossoma2*
`public Vector` **obterCromossoma2**`( )`
  - **Usage**
    * Permite obter o cromossoma actual
  - **Returns** - Vector de inteiros que representam o cromossoma actual

---

- *obterCusto*
`public int` **obterCusto**`( )`
  - **Usage**
    * Permite obter o custo do cromossoma
  - **Returns** - Custo do cromossoma

- *obterNumeroDeAlelos*
`public int` **obterNumeroDeAlelos**`( )`
  - **Usage**
    * Permite obter o número de alelos no cromossoma
  - **Returns** - Número de alelos no cromossoma

- *toString*
`public String` **toString**`( )`
  - **Usage**
    * Transforma o cromossoma numa string
  - **Returns** - String com o cromossoma

2.1.4 Class **Cruzamento**

Esta classe implementa mecanismos de cruzamentos de cromossomas codificados por sequências de Arestas.

Declaration

public class Cruzamento
extends java.lang.Object

CONSTRUCTORS

- *Cruzamento*
  `public` **Cruzamento**`( )`

METHODS

- *ArestasCrossoverASCM*
  `public static Vector` **ArestasCrossoverASCM**`( java.util.Vector` **seleccao**`, pc.dissertacao.ua.grafos.Grafo` **grafo** `)`
  - **Usage**
    - Permite efectuar a operação de cruzamento entre pares de cromossomas codificados por sequências de Arestas, para árvores de suporte de custo mínimo sem restrições, onde se procede à sua junção do par de cromossomas e depois é aplicado o algoritmo PrimRST
  - **Parameters**
    - `seleccao` - Vector de cromossomas seleccionados
    - `grafo` - Problema a optimizar
  - **Returns** - Vector de cromossomas cruzados admissíveis

- *ArestasCrossoverASCMRSalto*
  `public static Vector` **ArestasCrossoverASCMRSalto**`( java.util.Vector` **seleccao**`, pc.dissertacao.ua.grafos.Grafo` **grafo**`, int` **IDVerticeRaiz**`, int` **numeroMaximoDeSaltos** `)`
  - **Usage**
    - Permite efectuar a operação de cruzamento entre pares de cromossomas codificados por sequências de Arestas, para árvores de suporte de custo mínimo com restrições de salto, onde se procede à sua junção do par de cromossomas e depois é aplicado o algoritmo PrimRST
  - **Parameters**
    - `seleccao` - Vector de cromossomas seleccionados
    - `grafo` - Problema a optimizar
    - `IDVerticeRaiz` - Número que identifica o vértice raiz
    - `numeroMaximoDeSaltos` - Número máximo de saltos entre o vértice raiz e qualquer outro vértice da árvore de suporte de custo mínimo
  - **Returns** - Vector de cromossomas cruzados admissíveis

2.1.5 CLASS MotorEvolutivo

Esta classe permite executar o algoritmo evolutivo durante as várias gerações/iterações requeridas. Permite também executar testes (várias execuções do algoritmo evolutivo) e guardar os resultados num ficheiro.

DECLARATION

public class MotorEvolutivo **extends** java.lang.Object

FIELDS

- public static final int ASCM
  - Constante que representa o problema da árvore de suporte de custo mínimo sem restrições
- public static final int ASCMRSalto
  - Constante que representa o problema da árvore de suporte de custo mínimo com restrições de salto

CONSTRUCTORS

- *MotorEvolutivo*
  `public` **MotorEvolutivo**`( )`

METHODS

- *configurar*
  `public void` **configurar**`( int` **tipoASCM**`, int` **numIteracoes**`, int` **dimPopulacao**`, pc.dissertacao.ua.grafos.Grafo` **grafo**`, int` **objectivo**`, int` **dimensaoTorneio**`, int` **numFinalistasTorneio**`, int` **percentagemMutacao**`, int` **numIteracoesParaRenovacaoDaPopulacao**`, int` **numIteracoesParaImprimirResultado**`, boolean` **guardarTodasIteracoes** `)`
  - **Usage**
    - Permite configurar os parâmetros do motor evolutivo.
  - **Parameters**
    - `tipoASCM` - Tipo de árvore suporte de custo mínimo a ser usada no algoritmo evolutivo (ver constantes acima)
    - `numIteracoes` - Número de iterações do algoritmo genético
    - `dimPopulacao` - Número de elementos da população do algoritmo genético
    - `grafo` - Problema a optimizar
    - `objectivo` - Permite definir se queremos minimizar ou maximizar a função objectivo/aptidão
    - `dimensaoTorneio` - Número de elementos da população que são escolhidos ao acaso
    - `numFinalistasTorneio` - Número de elementos finalistas do torneio (mínimo são 2 pais)
    - `percentagemMutacao` - Percentagem de mutação para os elementos cruzados.
    - `numIteracoesParaRenovacaoDaPopulacao` - Número de iterações para renovação da população
    - `numIteracoesParaImprimirResultado` - Número de iterações para imprimir resultado
    - `guardarTodasIteracoes` - Permite guardar valores em todas as iterações
- *configurarASCMRSalto*
  `public void` **configurarASCMRSalto**`( pc.dissertacao.ua.grafos.Grafo` **grafo**`, int` **numeroMaximoDeSaltos**`, boolean` **usarHeuristicaGeracaoPopulacao** `)`
  - **Usage**
    - Permite configurar o algoritmo evolutivo para o problema da árvore de suporte de custo mínimo com restrições de salto (o vértice raiz é o último vértice do grafo)
  - **Parameters**

  * `grafo` - Problema a optimizar
  * `numeroMaximoDeSaltos` - Número máximo de saltos entre o vértice raiz e qualquer outro vértice da árvore de suporte de custo mínimo
  * `usarHeuristicaGeracaoPopulacao` - Usar heurística para geração da população

- *configurarASCMRSalto*
  `public void` **configurarASCMRSalto**`( pc.dissertacao.ua.grafos.Grafo` **grafo**`, int` **IDVerticeRaiz**`, int` **numeroMaximoDeSaltos**`, boolean` **usarHeuristicaGeracaoPopulacao** `)`

  – **Usage**
    * Permite configurar o algoritmo evolutivo para o problema da árvore de suporte de custo mínimo com restrições de salto, com a definição manual do vértice raiz
  – **Parameters**
    * `grafo` - Problema a optimizar
    * `IDVerticeRaiz` - Número que identifica o vértice raiz
    * `numeroMaximoDeSaltos` - Número máximo de saltos entre o vértice raiz e qualquer outro vértice da árvore de suporte de custo mínimo
    * `usarHeuristicaGeracaoPopulacao` - Usar heurística para geração da população

- *executar*
  `public void` **executar**`( )`

  – **Usage**
    * Permite executar o algoritmo evolutivo com a configuração definida na criação do motor evolutivo

---

- *testarCSV*
  `public void` **testarCSV**`( int` **numeroDeTestes**`, java.lang.String` **nomeFicheiroResultados** `)`

  – **Usage**
    * Permite executar um determinado número de vezes um algoritmo evolutivo com as mesmas configurações e guardar os resultados de cada execução num ficheiro CSV.
  – **Parameters**
    * `numeroDeTestes` - Número de testes a efectuar
    * `nomeFicheiroResultados` - Nome do ficheiro de resultados

- *testarExportarExcel*
  `public void` **testarExportarExcel**`( int` **numeroDeTestes**`, java.lang.String` **nomeFicheiroResultados**`, java.lang.String` **nomeFolhaExcel**`, int` **IDTeste** `)`

  – **Usage**
    * Permite executar um determinado número de vezes um algoritmo evolutivo com as mesmas configurações e guardar os resultados de cada execução num ficheiro CSV.
  – **Parameters**
    * `numeroDeTestes` - Número de testes a efectuar
    * `nomeFicheiroResultados` - Nome do ficheiro de resultados
    * `nomeFolhaExcel` - Nome da folha excel para guardar os resultados
    * `IDTeste` - Indica qual o teste que está a ser efectuado

2.1.6 Class **Mutacao**

Esta classe implementa mecanismos de mutação de cromossoma codificados por sequências de Arestas.

Declaration

public class Mutacao
extends java.lang.Object

Constructors

- *Mutacao*
  `public` **Mutacao**`( )`

Methods

- *MutacaoArestasASCM*
  `public static Vector` **MutacaoArestasASCM**`( java.util.Vector` **seleccao**`, pc.dissertacao.ua.grafos.Grafo` **grafo**`, int` **percentagemMutacao** `)`
  - **Usage**
    - Permite efectuar a mutação de cromossomas codificados por sequências de Arestas para o problema da árvore de suporte de custo mínimo sem restrições. É escolhido aleatoriamente um índice para sofrer a mutação.
  - **Parameters**
    - `seleccao` - Vector de cromossomas cruzados
    - `grafo` - Problema a optimizar
    - `percentagemMutacao` - (0..100) indica a percentagem de cromossomas cruzados que sofrem uma mutação
  - **Returns** - Vector de cromossomas mutados admissíveis

2.1.7 Class **Populacao**

Esta classe implementa mecanismos que permitem gerar uma população.

Declaration

public class Populacao
extends java.lang.Object

CONSTRUCTORS

- *Populacao*
  `public` **Populacao(** `int` **tamanho,** `pc.dissertacao.ua.grafos.Grafo` **grafo)**
  - **Usage**
    - Permite gerar uma instância da classe população para o problema da árvore de suporte de custo mínimo sem restrições
  - **Parameters**
    - `tamanho` - Número de elementos na população
    - `grafo` - Problema a optimizar

- *Populacao*
  `public` **Populacao(** `int` **tamanho,** `pc.dissertacao.ua.grafos.Grafo` **grafo,** `int` **IDVerticeRaiz,** `int` **numeroMaximoDeSaltos,** `boolean` **usarHeuristicaGeracaoPopulacao)**
  - **Usage**
    - Permite gerar uma instância da classe população para o problema da árvore de suporte de custo mínimo com restrições de salto
  - **Parameters**
    - `tamanho` - Número de elementos na população
    - `grafo` - Problema a optimizar
    - `IDVerticeRaiz` - Número que identifica o vértice raiz
    - `numeroMaximoDeSaltos` - Número máximo de saltos entre o vértice raiz e qualquer outro vértice da árvore de suporte de custo mínimo
    - `usarHeuristicaGeracaoPopulacao` - Usar heurística para geração da população

METHODS

- *gerarPopulacaoArestasASCM*
  `public void` **gerarPopulacaoArestasASCM(** `int` **dimensao,** `pc.dissertacao.ua.grafos.Grafo` **grafo)**
  - **Usage**
    - Permite gerar uma população para o problema da árvore de suporte de custo mínimo sem restrições. Nota 1: A população pode ter elementos cujo valor do cromossoma é igual. Nota 2: Todos os elementos da população são admissíveis para o problema
  - **Parameters**
    - `dimensao` - Número de elementos a gerar para a população
    - `grafo` - Problema a optimizar

- *gerarPopulacaoArestasASCMRSalto*
  `public void` **gerarPopulacaoArestasASCMRSalto(** `int` **dimensao,** `pc.dissertacao.ua.grafos.Grafo` **grafo,** `int` **IDVerticeRaiz,** `int` **numeroMaximoDeSaltos,** `boolean` **usarHeuristicaGeracaoPopulacao)**
  - **Usage**
    - Permite gerar uma população para o problema da árvore de suporte de custo mínimo com restrições de salto.
  - **Parameters**

    * `dimensao` - Número de elementos a gerar para a população
    * `grafo` - Problema a optimizar
    * `IDVerticeRaiz` - Número que identifica o vértice raiz
    * `numeroMaximoDeSaltos` - Número máximo de saltos entre o vértice raiz e qualquer outro vértice da árvore de suporte de custo mínimo
    * `usarHeuristicaGeracaoPopulacao` - Usar heurística para geração da população

- *imprimirPopulacao*
  `public void` **imprimirPopulacao**`( )`
  - **Usage**
    * Permite imprimir a população com o respectivo custo

- *obterCromossoma*
  `public Cromossoma` **obterCromossoma**`( int` **index** `)`
  - **Usage**
    * Permite obter um elemento (cromossoma) da população através da sua posição no vector de cromossomas
  - **Parameters**
    * `index` - Posição no vector de cromossomas
  - **Returns** - Instância da classe Cromossoma

- *obterDimensaoPopulacao*
  `public int` **obterDimensaoPopulacao**`( )`
  - **Usage**
    * Permite obter o número de elementos da população
  - **Returns** - Número de elementos da população

- *obterMaisApto*
  `public Cromossoma` **obterMaisApto**`( int` **MINMAX** `)`
  - **Usage**
    * Permite obter o elemento mais apto da população
  - **Parameters**
    * `MINMAX` - Indica se queremos Maximizar ou Minimizar a função objectivo
  - **Returns** - Elemento mais apto da população

- *renovarPopulacaoASCM*
  `public void` **renovarPopulacaoASCM**`( int` **dimensao**`,`
  `pc.dissertacao.ua.grafos.Grafo` **grafo**`, int` **MINMAX** `)`
  - **Usage**
    * Permite renovar a população para o problema da árvore de suporte de custo mínimo sem restrições, no qual o melhor elemento permanece na população e os restantes são gerados.
  - **Parameters**
    * `dimensao` - Número de elementos a gerar para a população
    * `grafo` - Problema a optimizar
    * `MINMAX` - Indica se queremos Maximizar ou Minimizar a função objectivo

---

- *renovarPopulacaoASCMRSalto*
  `public void` **renovarPopulacaoASCMRSalto**`( int` **dimensao**`, pc.dissertacao.ua.grafos.Grafo` **grafo**`, int` **MINMAX**`, int` **IDVerticeRaiz**`, int` **numeroMaximoDeSaltos**`, boolean` **usarHeuristicaGeracaoPopulacao** `)`

  – **Usage**
    * Permite renovar a população para o problema da árvore de suporte de custo mínimo com restrições de salto, no qual o melhor elemento permanece na população e os restantes são gerados.
  – **Parameters**
    * `dimensao` - Número de elementos a gerar para a população
    * `grafo` - Problema a optimizar
    * `MINMAX` - Indica se queremos Maximizar ou Minimizar a função objectivo
    * `IDVerticeRaiz` - Número que identifica o vértice raiz
    * `numeroMaximoDeSaltos` - Número máximo de saltos entre o vértice raiz e qualquer outro vértice da árvore de suporte de custo mínimo
    * `usarHeuristicaGeracaoPopulacao` - Usar heurística para renovação da população

- *substituirMenosAptos*
  `public void` **substituirMenosAptos**`( java.util.Vector` **cromossomas**`, int` **MINMAX** `)`

  – **Usage**
    * Permite substituir os elementos menos aptos da população. Nota: Os elementos menos aptos são sempre substituídos pelos novos elementos mesmo que a aptidão destes seja inferior. Isto favorece a heterogeneidade da população.
  – **Parameters**
    * `cromossomas` - Vector com os novos cromossomas
    * `MINMAX` - Indica se queremos Maximizar ou Minimizar a função objectivo

2.1.8 Class **Seleccao**

Esta classe implementa mecanismos que efectuam a selecção por torneio dos elementos da população.

Declaration

public class Seleccao
extends java.lang.Object

Constructors

- *Seleccao*
  `public` **Seleccao**`( )`

Methods

- *seleccaoPorTorneio*
  `public static Vector` **seleccaoPorTorneio**`( pc.dissertacao.ua.algoevo.Populacao` **populacao**`, int` **dimensaoTorneio**`, int` **numFinalistasTorneio**`, int` **MINMAX** `)`
  - **Usage**
    - Permite obter um vector de cromossomas seleccionados por torneio
  - **Parameters**
    - `populacao` - População sobre o qual se processa o torneio
    - `dimensaoTorneio` - Número de elementos a seleccionar aleatoriamente da população
    - `numFinalistasTorneio` - Número de elementos com maior aptidão que constituem os seleccionados do torneio (deve ser inferior ou igual a numFinalistasTorneio)
    - `MINMAX` - Indica se pretendemos maximizar ou minimizar a função objectivo/aptidão
  - **Returns** - Vector de cromossomas seleccionados

Capítulo 3

Package pc.dissertacao.ua.grafos

Package Contents *Page*

Classes

3.1 Classes

3.1.1 Class **Aresta**

Esta classe implementa mecanismos que definem a aresta de um grafo.

Declaration

public class Aresta
extends java.lang.Object

Constructors

- *Aresta*
  `public` **Aresta**`( int` **id** `)`
  - **Usage**
    - Cria uma instância da classe Aresta com id definido
  - **Parameters**
    - `id` - Número associado à aresta

- *Aresta*
  `public` **Aresta**`( int` **id**`, int` **custo** `)`
  - **Usage**
    - Cria uma instância da classe Aresta com id e custo definidos
  - **Parameters**
    - `id` - Número associado à aresta
    - `custo` - Custo associado à aresta

Methods

- *imprimirAresta*
  `public void` **imprimirAresta**`( )`
  - **Usage**
    - Permite imprimir a aresta

- *obterCusto*
  `public int` **obterCusto**`( )`
  - **Usage**
    - Permite obter o custo da aresta
  - **Returns** - Custo associado à aresta

- *obterID*
  `public int` **obterID**`( )`

  - **Usage**
    * Permite obter o id da aresta
  - **Returns** - Número associado à aresta

- *toString*
  `public String` **toString**`( )`
  - **Usage**
    * Transforma o id da aresta numa string
  - **Returns** - String com o id da aresta

3.1.2 Class Codificacao

Esta classe implementa mecanismos para a codificação e descodificação de sequências e cromossomas. Está implementada a codificação Arestas.

Declaration

public class Codificacao **extends** java.lang.Object

Constructors

- *Codificacao*
  `public` **Codificacao**`( )`

Methods

- *codificarCromossomaArestas*
  `public static Cromossoma` **codificarCromossomaArestas**`(`
  `pc.dissertacao.ua.grafos.Grafo` **arvore** `)`
  - **Usage**
    * Permite codificar uma árvore numa sequência de Arestas
  - **Parameters**
    * `arvore` - Árvore de suporte a codificar
  - **Returns** - Instância da classe Cromossoma que representa a sequência de Arestas

---

- *codificarSequenciaArestas*
  `public static Vector` **codificarSequenciaArestas**`( pc.dissertacao.ua.grafos.Grafo`
  arvore `)`
  - **Usage**
    * Permite codificar uma árvore numa sequência de Arestas
  - **Parameters**
    * `arvore` - Árvore de suporte a codificar
  - **Returns** - Vector de pares de vértices que representa a sequência de Arestas

---

- *descodificarCromossomaArestas*
  `public static Grafo` **descodificarCromossomaArestas**`(`
  `pc.dissertacao.ua.algoevo.Cromossoma` **cromossoma** `)`
  - **Usage**
    - Permite descodificar um cromossoma de Arestas numa árvore
  - **Parameters**
    - `cromossoma` - Instância da classe Cromossoma que representa a sequência de Arestas
  - **Returns** - Árvore de suporte descodificada

- *descodificarSequenciaArestas*
  `public static Grafo` **descodificarSequenciaArestas**`( java.util.Vector` **seqArestas** `)`
  - **Usage**
    - Permite descodificar uma sequência de Arestas numa árvore
  - **Parameters**
    - `seqArestas` - Vector de pares de vértices que representa a sequência de Arestas
  - **Returns** - Árvore de suporte descodificada

- *descodificarSequenciaPrufer*
  `public static Grafo` **descodificarSequenciaPrufer**`( java.util.Vector` **prufer** `)`
  - **Usage**
    - Permite descodificar uma sequência de Prüfer numa árvore
  - **Parameters**
    - `prufer` - Vector de inteiros que representa a sequência de Prüfer
  - **Returns** - Árvore de suporte descodificada

- *imprimirSequenciaArestas*
  `public static void` **imprimirSequenciaArestas**`( java.util.Vector` **seqArestas** `)`
  - **Usage**
    - Permite imprimir uma sequência de Arestas
  - **Parameters**
    - `seqArestas` -

- *juntarCromossomas*
  `public static Cromossoma` **juntarCromossomas**`(`
  `pc.dissertacao.ua.algoevo.Cromossoma` **cromossoma1**`,`
  `pc.dissertacao.ua.algoevo.Cromossoma` **cromossoma2** `)`
  - **Usage**
    - Permite juntar dois cromossomas de Arestas (para efectuar o cruzamento de dois cromossomas de Arestas)
  - **Parameters**
    - `cromossoma1` - Cromossoma que codifica uma sequência de Arestas
    - `cromossoma2` - Cromossoma que codifica uma sequência de Arestas
  - **Returns** - Cromossoma que codifica uma sequência de Arestas comuns aos dois cromossomas parâmetros

3.1.3 Class **Grafo**

Esta classe implementa mecanismos de criação e trabalho com grafos, através de mapas de adjacência e de incidência.

Declaration

public class Grafo
extends java.lang.Object

Fields

- public Map mapaAdjacencia
  - Variável que representa o mapa de adjacências (IDVertice, Set)
- public Map mapaIncidencia
  - Variável que representa o mapa de incidências (IDAresta, ParDeVertices)
- public SortedMap vertices
  - Variável que representa o mapa de vértices (IDVertice, Vertice)
- public SortedMap arestas
  - Variável que representa o mapa de arestas (IDAresta, Aresta)

Constructors

- *Grafo*
  `public` **Grafo**`( )`
  - **Usage**
    - Cria uma instância da classe Grafo

- *Grafo*
  `public` **Grafo**`( java.util.Map` **mapaAdjacencia**`, java.util.Map` **mapaIncidencia**`, java.util.SortedMap` **vertices**`, java.util.SortedMap` **arestas** `)`
  - **Usage**
    - Cria uma instância da classe Grafo com parâmetros explícitos
  - **Parameters**
    - `mapaAdjacencia` - Mapa com as adjacências vértice-aresta (IDVertice, Set)
    - `mapaIncidencia` - Mapa com as incidências aresta-vértices (IDAresta, ParDeVertices)
    - `vertices` - Mapa de vértices (IDVertice, Vertice)
    - `arestas` - Mapa de arestas (IDAresta, Aresta)

METHODS

- *actualizarSaltos*
  `public void` **actualizarSaltos**`( int` **IDVerticeRaiz** `)`
  - **Usage**
    - Permite actualizar os saltos dos vértices de uma árvore de suporte de custo mínimo. (Deve apenas ser usado com instâncias desta classe que sejam árvores)
  - **Parameters**
    - `IDVerticeRaiz` - Número que identifica o vértice raiz

- *adicionarAresta*
  `public boolean` **adicionarAresta**`( int` **IDVertice1**`, int` **IDVertice2** `)`
  - **Usage**
    - Permite adicionar uma aresta ao grafo
  - **Parameters**
    - `IDVertice1` - Número que identifica um vértice extremo
    - `IDVertice2` - Número que identifica outro vértice extremo
  - **Returns** - Verdadeiro se a aresta foi adicionada ao grafo

---

- *adicionarAresta*
  `public boolean` **adicionarAresta**`( int` **IDVertice1**`, int` **IDVertice2**`, int` **custo** `)`
  - **Usage**
    - Permite adicionar uma aresta ao grafo
  - **Parameters**
    - `IDVertice1` - Número que identifica um vértice extremo
    - `IDVertice2` - Número que identifica outro vértice extremo
    - `custo` - Custo associado a aresta
  - **Returns** - Verdadeiro se a aresta foi adicionada ao grafo

---

- *adicionarVertice*
  `public boolean` **adicionarVertice**`( int` **IDVertice** `)`
  - **Usage**
    - Permite adicionar um vértice ao grafo
  - **Parameters**
    - `IDVertice` - Número do vértice a adicionar
  - **Returns** - Verdadeiro se o vértice foi adicionado ao grafo

---

- *clonarGrafo*
  `public Grafo` **clonarGrafo**`( )`
  - **Usage**
    - Permite obter uma cópia do grafo actual
  - **Returns** - Uma instância da classe Grafo com os mesmos valores do grafo actual

---

- *existeAresta*
  `public boolean` **existeAresta**`( pc.dissertacao.ua.grafos.Aresta` **aresta** `)`

  - **Usage**
    * Permite verificar se uma aresta existe no grafo
  - **Parameters**
    * `aresta` - Instância da classe Aresta a verificar
  - **Returns** - Verdadeiro se essa instância existe no grafo

---

- *existeAresta*
`public boolean` **existeAresta**`( int` **IDAresta** `)`
  - **Usage**
    * Permite verificar se uma aresta existe no grafo
  - **Parameters**
    * `IDAresta` - Número que identifica a aresta
  - **Returns** - Verdadeiro se existe no grafo uma aresta com esse id

---

- *existeAresta*
`public boolean` **existeAresta**`( int` **IDVertice1**`, int` **IDVertice2** `)`
  - **Usage**
    * Permite verificar se uma aresta existe no grafo
  - **Parameters**
    * `IDVertice1` - Número que identifica um vértice extremo
    * `IDVertice2` - Número que identifica outro vértice extremo
  - **Returns** - Verdadeiro se existe no grafo uma aresta com esses vértices extremos

---

- *existeVertice*
`public boolean` **existeVertice**`( int` **IDVertice** `)`
  - **Usage**
    * Permite verificar se um vértice existe no grafo
  - **Parameters**
    * `IDVertice` - Número que identifica o vértice
  - **Returns** - Verdadeiro se existe no grafo um vértice com esse id

---

- *existeVertice*
`public boolean` **existeVertice**`( pc.dissertacao.ua.grafos.Vertice` **vertice** `)`
  - **Usage**
    * Permite verificar se um vértice existe no grafo
  - **Parameters**
    * `vertice` - Instância da classe Vertice a verificar
  - **Returns** - Verdadeiro se essa instância existe no grafo

---

- *gerarCromossomaArestasASCM*
`public Cromossoma` **gerarCromossomaArestasASCM**`( )`
  - **Usage**
    * Permite gerar uma instância de Cromossoma para a árvore de suporte de custo mínimo com restrições que é admissível para o grafo actual
  - **Returns** - Uma instância de Cromossoma que é admissível para o grafo actual

---

- *gerarCromossomaArestasASCMRSalto*
  `public Cromossoma` **gerarCromossomaArestasASCMRSalto**`( int` **IDVerticeRaiz**`, int` **numeroMaximoDeSaltos** `)`
  - **Usage**
    - Permite gerar aleatoriamente uma instância de Cromossoma para a árvore de suporte de custo mínimo com restrições de salto que é admissível para o grafo actual
  - **Parameters**
    - `IDVerticeRaiz` - Número que identifica o vértice raiz
    - `numeroMaximoDeSaltos` - Número máximo de saltos entre o vértice raiz e qualquer outro vértice da árvore de suporte de custo mínimo
  - **Returns** - Uma instância de Cromossoma que é admissível para o grafo actual

---

- *gerarCromossomaArestasASCMRSaltoHeuristica*
  `public Cromossoma` **gerarCromossomaArestasASCMRSaltoHeuristica**`( int` **IDVerticeRaiz**`, int` **numeroMaximoDeSaltos** `)`
  - **Usage**
    - Permite gerar uma instância de Cromossoma para a árvore de suporte de custo mínimo com restrições de salto que é admissível para o grafo actual (utiliza a heurística para a geração/renovação da população)
  - **Parameters**
    - `IDVerticeRaiz` - Número que identifica o vértice raiz
    - `numeroMaximoDeSaltos` - Número máximo de saltos entre o vértice raiz e qualquer outro vértice da árvore de suporte de custo mínimo
  - **Returns** - Uma instância de Cromossoma que é admissível para o grafo actual

---

- *imprimirAdjacencias*
  `public void` **imprimirAdjacencias**`( )`
  - **Usage**
    - Permite imprimir a lista de adjacências do grafo

- *imprimirArestas*
  `public void` **imprimirArestas**`( )`
  - **Usage**
    - Permite imprimir a lista das arestas do grafo

- *imprimirIncidencias*
  `public void` **imprimirIncidencias**`( )`
  - **Usage**
    - Permite imprimir a lista de incidências do grafo

- *imprimirVertices*
  `public void` **imprimirVertices**`( )`
  - **Usage**
    - Permite imprimir a lista dos vértices do grafo

- *imprimirVerticesESaltos*
  `public void` **imprimirVerticesESaltos**`( )`

  - **Usage**
    - Permite imprimir a lista dos vértices (e saltos) do grafo

---

- *obterAresta*
`public Aresta` **obterAresta**`( int` **IDVertice1**`, int` **IDVertice2** `)`
  - **Usage**
    - Permite obter uma aresta existente no grafo
  - **Parameters**
    - `IDVertice1` - Número que identifica um vértice extremo
    - `IDVertice2` - Número que identifica outro vértice extremo
  - **Returns** - Null se a aresta não existe, caso contrário retorna uma instância da classe Aresta

---

- *obterArestas*
`public Collection` **obterArestas**`( )`
  - **Usage**
    - Permite obter a lista de aresta que estão no grafo
  - **Returns** - Lista de aresta que estão no grafo

- *obterIDVerticeVizinho*
`public int` **obterIDVerticeVizinho**`( int` **IDVertice** `)`
  - **Usage**
    - Permite obter o id de um vértice vizinho de um determinado vértice (Usar sempre em conjunto com obterNumeroDeVerticesVizinhos)
  - **Parameters**
    - `IDVertice` - Número que identifica o vértice
  - **Returns** - Id de um vértice vizinho desse vértice

- *obterIncidencia*
`public ParDeVertices` **obterIncidencia**`( int` **IDAresta** `)`
  - **Usage**
    - Permite obter o par de vértices incidente numa determinada aresta
  - **Parameters**
    - `IDAresta` - Número da aresta a obter os vértices incidentes
  - **Returns** - Par de vértices incidente na aresta

- *obterNumeroDeArestas*
`public int` **obterNumeroDeArestas**`( )`
  - **Usage**
    - Permite obter o número de arestas no grafo
  - **Returns** - Número de arestas no grafo

- *obterNumeroDeVertices*
`public int` **obterNumeroDeVertices**`( )`
  - **Usage**
    - Permite obter o número de vértices no grafo
  - **Returns** - Número de vértices no grafo

- *obterNumeroDeVerticesVizinhos*
`public int` **obterNumeroDeVerticesVizinhos**`( int` **IDVertice** `)`
  - **Usage**
    - Permite obter o número de vértices vizinhos de um determinado vértice
  - **Parameters**
    - `IDVertice` - Número que identifica o vértice
  - **Returns** - Número de vértices vizinhos desse vértice

- *obterParDeVertices*
`public Collection` **obterParDeVertices**`( )`
  - **Usage**
    - Permite obter os pares de vértices do mapa de incidência
  - **Returns** - Pares de vértices do mapa de incidência

- *obterVertices*
`public Collection` **obterVertices**`( )`
  - **Usage**
    - Permite obter a lista de vértices que estão no grafo
  - **Returns** - Lista de vértices que estão no grafo

- *removerTodosVertices*
`public void` **removerTodosVertices**`( )`
  - **Usage**
    - Permite remover todos os vértices do grafo

- *removerVertice*
`public boolean` **removerVertice**`( int` **IDVertice** `)`
  - **Usage**
    - Permite remover um vértice do grafo
  - **Parameters**
    - `IDVertice` - Número que identifica o vértice
  - **Returns** - Verdadeiro se o vértice foi removido do grafo

3.1.4 Class **ParDeVertices**

Esta classe permite agrupar 2 vértices para armazenar no mapa de incidências

Declaration

public class ParDeVertices
extends java.lang.Object

CONSTRUCTORS

- *ParDeVertices*
  `public` **ParDeVertices**`( int` **IDVertice1**`, int` **IDVertice2** `)`
  - **Usage**
    - Cria uma nova instância da classe
  - **Parameters**
    - `IDVertice1` - Número que identifica um vértice
    - `IDVertice2` - Número que identifica outro vértice

METHODS

- *imprimirParDeVertices*
  `public void` **imprimirParDeVertices**`( )`
  - **Usage**
    - Permite imprimir o par de vértices

- *obterIDVertice1*
  `public int` **obterIDVertice1**`( )`
  - **Usage**
    - Permite obter o id de um vértice pertencente ao par de vértices
  - **Returns** - Id de um vértice pertencente ao par de vértices

---

- *obterIDVertice2*
  `public int` **obterIDVertice2**`( )`
  - **Usage**
    - Permite obter o id de outro vértice pertencente ao par de vértices
  - **Returns** - Id de outro vértice pertencente ao par de vértices

---

- *toString*
  `public String` **toString**`( )`
  - **Usage**
    - Transforma o par de vértices numa string
  - **Returns** - String com os ids dos vértices

- *toString2*
  `public String` **toString2**`( )`
  - **Usage**
    - Transforma o par de vértices numa string (apenas o número dos vértices)
  - **Returns** - String com os ids dos vértices

3.1.5 CLASS **Vertice**

Esta classe implementa mecanismos que definem o vértice de um grafo.

Declaration

public class Vertice
extends java.lang.Object

Constructors

- *Vertice*
  `public` **Vertice(** `int` **id)**
  - **Usage**
    - Cria uma instância da classe Vertice com id definido
  - **Parameters**
    - `id` - Número associado ao vértice

Methods

- *definirSalto*
  `public boolean` **definirSalto(** `int` **salto)**
  - **Usage**
    - Permite definir o salto associado a este vértice em relação a um vértice raiz
  - **Parameters**
    - `salto` - Número de saltos até ao vértice raiz
  - **Returns** - Indica se o número de saltos foi adicionado com sucesso ao vértice

---

- *imprimirVertice*
  `public void` **imprimirVertice(** **)**
  - **Usage**
    - Permite imprimir o vértice

- *imprimirVerticeESalto*
  `public void` **imprimirVerticeESalto(** **)**
  - **Usage**
    - Permite imprimir o vértice e salto

- *obterID*
  `public int` **obterID(** **)**
  - **Usage**
    - Permite obter o id do vértice
  - **Returns** - Número associado ao vértice

- *obterSalto*
  `public int` **obterSalto(** **)**
  - **Usage**

    * Permite obter o número de saltos até ao vértice raiz
  - **Returns** - Número de saltos até ao vértice raiz

- *toString*
  `public String` **toString**`( )`
  - **Usage**
    * Transforma o id do vértice numa string
  - **Returns** - String com o id do vértice

Bibliografia

[1] R. K. Ahuja, T. L. Magnanti e J. B. Orlin. *Network Flows: Theory, Algorithms and Applications*. Prentice Hall, 1993. ISBN 9780136175490.

[2] T. Blickle e L. Thiele. A comparison of selection schemes used in genetic algorithms. *Evolutionary Computation*, **4(4)**:361–394, 1996.

[3] R. Cadenhead e L. Lemay. *Sams Teach Yourself Java 6 in 21 Days*. Sams, 2007. ISBN 9780672329432.

[4] D. M. Cardoso, J. Szymanaki e M. Rostami. *Matemática Discreta: Combinatória, Teoria dos Grafos, Algoritmos*. Escolar Editora, 2009. ISBN 9789725922378.

[5] D. A. Coley. *An Introduction to Genetic Algorithms for Scientists and Engineers*. World Scientific Publishing Company, 1997. ISBN 9789810236021.

[6] G. Dahl, L. Gouveia e C. Requejo. On formulations and methods for the hop-constrained minimum spanning tree problem. In M. G. C. Resende e P. M. Pardalos, editores, *Handbook of Optimization in Telecommunications*, 493–515. Springer Science + Business Media, 2006. ISBN 9780387306629.

[7] C. Darwin. *On the Origin of Species by Means of Natural Selection, or the Preservation of Favoured Races in the Struggle for Life*. John Murray, 1859.

[8] T. C. Fogarty. Adaptive rule-based optimisation of combustion in multiple burner installations. *Expert Systems in Engineering Principles and Applications, Lecture Notes in Computer Science*, **462**:241–248, 1990.

[9] M. Gen e R. Cheng. *Genetic Algorithms and Engineering Optimization*. Wiley-Interscience, 1999. ISBN 9780471315315.

[10] D. E. Goldberg. *Computer-Aided Gas Pipeline Operation using Genetic Algorithms and Rule Learning*. PhD thesis, University of Michigan, 1983.

[11] D. E. Goldberg. *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison-Wesley Professional, 1989. ISBN 0201157675.

[12] D. E. Goldberg e K. Deb. A comparative analysis of selection schemes used in genetic algorithms. In G. J. E. Rawlins, editor, *Foundations of Genetic Algorithms I*, 69–93. Morgan Kaufmann, 1991. ISBN 9781558601703.

[13] D. E. Goldberg e R. E. Smith. Nonstationary function optimization using genetic algorithms with dominance and diploidy. In J. J. Grefenstette, editor, *Proc. of the 2nd International Conference on Genetic Algorithms and Their Applications*, 59–68. 1987.

[14] J. F. Gonçalves e D. B. M. M. Fontes. A multi population genetic algorithm for hop-constrained trees in nonlinear cost flow networks. In *Proceedings of the International Network Optimization Conference INOC2009*. 2009. URL `http://www.liaad.up.pt/pub/2009/GF09/`.

[15] J. Gottlieb, B. A. Julstrom, G. R. Raidl e F. Rothlauf. Prüfer numbers: A poor representation of spanning trees for evolutionary search. In *Procedings of the Genetic and Evolutionary Computation Conference*, 343–350. 2001.

[16] L. Gouveia. Using the Miller-Tucker-Zemlin constraints to formulate a minimal spanning tree problem with hop constraints. *Computers and Operations Research*, **22(9)**: 959–970, 1995.

[17] L. Gouveia. Multicommodity flow models for spanning trees with hop constraints. *European Journal of Operational Research*, **95**:178–190, 1996.

[18] L. Gouveia. Using variable redefinition for computing lower bounds for minimum spanning and Steiner trees with hop constraints. *INFORMS Journal on Computing*, **10(2)**:180–188, 1998.

[19] L. Gouveia e C. Requejo. A new lagrangean relaxation approach for the hop-constrained minimum spanning tree problem. *European Journal of Operational Research*, **132(3)**:539–552, 2001.

[20] L. Gouveia, L. Simonetti e E. Uchoa. Modeling hop-constrained and diameter-constrained minimum spanning tree problems as Steiner tree problems over layered graphs. *Mathematical Programming*, 2009. doi: 10.1007/s10107-009-0297-2. Online First (June 15, 2009).

[21] L. Gouveia, A. Paias e D. Sharma. Restricted dynamic programming based neighborhoods for the hop-constrained minimum spanning tree problem. *Journal of Heuristics*, 2010. doi: 10.1007/s10732-009-9123-5. Online First (December 28, 2009).

[22] B. A. Julstrom. A permutation-coded evolutionary algorithm for the bounded-diameter minimum spanning tree problem. In A. Barry, editor, *2003 Genetic and Evolutionary Computation Conference Workshop Program GECCO2003*, 2–7. 2003.

[23] B. A. Julstrom e G. R. Raidl. A weighted coding in a genetic algorithm for the degree-constrained minimum spanning tree problem. In *Applied Computing 2000: Proceedings of the 2000 ACM Symposium on Applied Computing SAC2000*, **1**:440–445, 2000.

[24] D. Kang, H. Hashimoto e F. Harashima. Path generation for mobile robot navigation using genetic algorithm. *Proceedings of the 21st International Conference on Industrial Electronics, Control, and Instrumentation*, **1**:167–172, 1995.

[25] J. Knowles, D. Corne e M. Oates. A new evolutionary approach to the degree-constrained minimum spanning tree problem. *IEEE Transactions on Evolutionary Computation*, **4**:125–134, 1999.

[26] M. Krishnamoorthy, A. T. Ernst e Y. M. Sharaiha. Comparison of algorithms for the degree constrained minimum spanning tree. *Journal of Heuristics*, **7(6)**:587–611, 2001.

[27] K. L. Mak e Y. S. Wong. Design of integrated production-inventory-distribution systems using genetic algorithm. *Genetic Algorithms in Engineering Systems: Innovations and Applications GALESIA95*, **414**:454–460, 1995.

[28] M. Mitchell. *An Introduction to Genetic Algorithms*. The MIT Press, 1996. ISBN 9780262133166.

[29] A. Nijenhuis e H. S. Wilf. *Combinatorial Algorithms for Computers and Calculators*. Academic Press, 1978. ISBN 9780125192606.

[30] C. C. Palmer e A. Kershenbaum. Representing trees in genetic algorithms. In *Proceedings of the First IEEE Conference on Evolutionary Computation*, **1**:379–384, 1994.

[31] R. Poli, W. B. Langdon e N. F. McPhee. *A Field Guide to Genetic Programming*. Lulu Enterprises, 2008. ISBN 9781409200734.

[32] G. R. Raidl. An efficient evolutionary algorithm for the degree-constrained minimum spanning tree problem. In *Proceedings of the 2000 IEEE Congress on Evolutionary Computation*, **1**:104–111, 2000.

[33] G. R. Raidl e C. Drexel. A predecessor coding in an evolutionary algorithm for the capacitated minimum spanning tree problem. In *Late-Breaking-Papers Procedings of the 2000 Genetic and Evolutionary Computation Conference*, 309–316. 2000.

[34] G. R. Raidl e B. A. Julstrom. Edge-sets: An effective evolutionary coding of spanning trees. *IEEE Transactions on Evolutionary Computation*, **7(3)**:225–239, 2003.

[35] H. Sarbazi-Azad. Gnome sort. *Computing Science Glasgow*, **599**:4, 2000.

[36] V. Schnecke e O. Vornberger. Genetic design of vlsi-layouts. *In Genetic Algorithms in Engineering Systems: Innovations and Applications GALESIA95*, **414**:430–435, 1995.

[37] S. N. Sivanandam e S. N. Deepa. *Introduction to Genetic Algorithms*. Springer, 2007. ISBN 9783540731894.

[38] B. Y. Wu e K.-M. Chao. *Spanning Trees and Optimization Problems*. Chapman and Hall/CRC, 2004. ISBN 9781584884361.